Practical Compact Indexes for Top-k Document Retrieval

SIMON GOG, Karlsruhe Institute of Technology
ROBERTO KONOW, University of Chile and Universidad Diego Portales
GONZALO NAVARRO, University of Chile

We present a fast and compact index for top-k document retrieval on general string collections. That is, given a string pattern, the index returns the k documents where it appears most often. We adapt a linear-space and optimal-time theoretical solution, whose implementation poses various algorithm engineering challenges. While a naive implementation of the optimal solution is estimated to require around $80n$ bytes for a text collection of n symbols, our implementation requires $2.5n$–$3.0n$ bytes, text included, and answers queries within microseconds. This outperforms all the previous practical indexes by orders of magnitude; the only index using less space is hundreds of times slower. Our index can be built on collections of hundreds of gigabytes and on tokenized text collections.

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Compact Data Structure , Top-k document retrieval

1. INTRODUCTION

The task of finding relevant information in large text collections poses many challenges to developers and users of Information Retrieval (IR) systems [Büttcher et al. 2010; Croft et al. 2009; Baeza-Yates and Ribeiro-Neto 2011]. The core task of IR systems, and its most common application in search engines, is to return the k documents from the collection that best relate to user queries.

The most prominent challenges to build a useful IR system are (1) quality, (2) time, and (3) space. The quality problem boils down to defining a suitable scoring scheme. Score functions range from very simple (such as "term frequency", the number of times the query appears in the document) to very sophisticated ones. In many cases a simple score function is used to filter a few candidate documents and more sophisticated ranking is then computed on those [Büttcher et al. 2010; Liu 2009]. This is because of the second concern, time. Text collections are usually too large to admit a sequential query processing. Indexes are data structures built on the text to speed up queries, and this is connected with the third challenge: space. Designing and implementing an index that provides a good trade-off in terms of space and time is a challenging problem from both a theoretical and a practical point of view.

In most search engines, top-k queries are solved using a well-known data structure called an *Inverted Index*. This is an old and simple, yet efficient, data structure that plays a central role in IR. Inverted Indexes have been designed for scenarios where queryable terms are predefined and are not too many compared to the size of the collection. For example, they work very well on collections where the documents can be

Funded with basal funds FB0001, Conicyt, Chile; with Fondecyt Grant 1-140796, Chile; and with a Conicyt PhD Scholarship.
Preliminary partial versions of this work appeared in Konow and Navarro [2013] and Gog and Navarro [2015].
Author's addresses: Simon Gog, Karlsruhe Institute of Technology (KIT), Germany, simon.gog@googlemail.com; Roberto Konow and Gonzalo Navarro, Center for Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile, rkonow@dcc.uchile.cl, gnavarro@dcc.uchile.cl.

DOI: 0000001.0000001

easily tokenized into "words", and queries are also formed by words. This is a common scenario for documents written in most Western languages. On the other hand, there are several applications where top-k queries are of interest but inverted indexes are not useful. One are the text collections written in East Asian languages such as Chinese or Korean, where words are difficult to segment, or in agglutinating languages such as Finnish and German, where one needs to search for particles inside words. There are also sequence collections with no concept of words, such as software source repositories, MIDI streams, DNA and protein sequences. In these cases, virtually any text substring is a potential query.

In this paper we focus on indexing a collection of generic *strings* to support top-k queries on it. The top-k document retrieval problem can then be defined as follows: a collection $\mathcal{D}$ of D documents, consisting of sequences of total length n over an alphabet of size σ, is preprocessed so that given a query string P of length m, the system retrieves k documents with the highest "score" to P, for some definition of score.

The basic solutions to this general problem build a *suffix tree* [Weiner 1973] or a *suffix array* [Manber and Myers 1993], which are indexes that can count and list all the individual occurrences of P in the collection, in optimal or near-optimal time. Still this functionality is not sufficient to solve top-k document retrieval efficiently. The problem of finding top-k documents containing the pattern as a substring, even with a simple relevance measure like term frequency, is challenging. Hon et al. [2010] presented the first efficient solution, achieving $O(m + \log n \log\log n + k)$ time, yet with superlinear space usage, $O(n \log^2 n)$ bits. Then Hon et al. [2009] improved the solution to $O(m + k \log k)$ time and linear space, $O(n \log n)$ bits. The problem was then essentially closed by Navarro and Nekrich [2012], who achieved optimal $O(m + k)$ time using $O(n(\log \sigma + \log D))$ bits.

In practice, however, those solutions are far from satisfactory. Implemented directly from the theory, the constants involved in the optimal (and previous) solutions are not small, especially in space: The index can be as large as 80 times the size of the collection, making it unfeasible in practice. Just a component of the solution, the suffix tree, can be 20 times larger than the text, and the suffix array, 4 times. There has also been a line of research aiming at optimal space, see Navarro [2014]. While they achieved important theoretical results, a verbatim implementation is likely to be equally insatisfactory.

The most practical implementations [Culpepper et al. 2010; Navarro et al. 2014b; Gog et al. 2014] have followed an intermediate path in terms of space, using 2–5 times the text size and answering queries within milliseconds. Still, they are toy implementations, limited to relatively small collections of a few hundred megabytes. They are also restricted to sequences over small alphabets (usually $\sigma = 255$ distinct symbols), which prevents using them on sequences of words, for example.

In this work, we show that a carefully engineered implementation of the optimal-time proposal [Navarro and Nekrich 2012] is competitive in space with current implementations and performs orders of magnitude faster. Our new top-k index implementation uses 2.5–3.0 times the text size and answers queries within microseconds, in collections of hundreds of gigabytes and over alphabets of up to millions of symbols. Our experimental comparison shows that our index is orders of magnitude faster than previous heuristics [Culpepper et al. 2010], naive solutions [Gog et al. 2014], and compressed solutions [Navarro et al. 2014b], whereas only the latter uses less space than ours. We also present results on collections of 500 GB, where existing implementations of the previous structures cannot be built.

Our ability to handle large alphabets allows us to apply our index on collections of natural language text, which are regarded as sequences of words (not characters), so

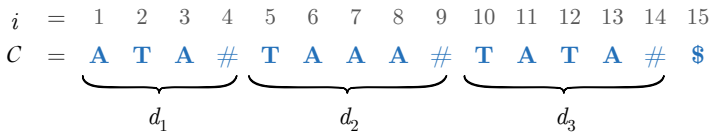


Fig. 1. Concatenation $\mathcal{C}$ of our 3-document example collection $\mathcal{D}$.

that our index offers a functionality similar to an inverted index. In this case our index takes about the same space of the tokenized collection (i.e., one integer per word, and this includes the storage of the collection itself).

This paper is structured as follows. Section 2 presents basic concepts. Section 3 provides a discussion on the state of the art and Section 4 describes the optimal-time top-k structures we implement. Section 5 describes a basic compact index that implements the theoretical proposal within reasonable space. Section 6 discusses various algorithmic improvements over the basic solution, obtaining improved time and space. Section 7 describes the experimental framework, collections and implementation details of the introduced indexes. Section 8 shows the space and time results for the new and previous indexes. Finally, Section 9 summarizes our results and discusses future paths to improve compact top-k indexes.

2. BASIC CONCEPTS

This section describes the basic concepts and data structures that are employed in the design and implementation of indexes for solving top-k document retrieval queries. Each one is described at the depth that is needed to follow the article.

2.1. Document collections

We define a collection of D documents $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$ containing symbols from an alphabet Σ of size σ, where each d_i ends with a special symbol $\# \in \Sigma$. We call $\mathcal{C} = d_1 d_2 \ldots d_D\$$ their concatenation, ended by another symbol $\$ \in \Sigma$; for the lexicographic comparisons we assume it holds $\$ < \# < c$ for any other $c \in \Sigma$. We call $n = |\mathcal{C}| = 1 + \sum_{i=1}^{D} |d_i|$. Fig. 1 shows an example of a collection containing 3 documents, of total length $n = 14$.

2.2. Suffix arrays

A *suffix array* [Manber and Myers 1993] is a common full-text index that allows us to efficiently retrieve, for an arbitrary pattern, the amount of its occurrences and their positions in a given text. Let $\mathrm{T} = t_1 t_2 \ldots t_n$ be a text, with $t_i \in \Sigma$ for all $1 \leq i < n$ and $t_n = \$$. The suffix array $\mathrm{SA}[1, n]$ contains pointers to every suffix of T, lexicographically sorted. For a position $i \in [1, n]$, $\mathrm{SA}[i]$ points to the suffix $\mathrm{T}[\mathrm{SA}[i], n] = t_{\mathrm{SA}[i]} t_{\mathrm{SA}[i]+1} \ldots t_n$, and it holds that $\mathrm{T}[\mathrm{SA}[i], n]$ is lexicographically smaller than $\mathrm{T}[\mathrm{SA}[i+1], n]$. To find the occurrences of an arbitrary pattern P of length m, two binary searches are performed to find the maximal interval $[sp, ep]$ such that for every position in $\mathrm{SA}[sp \leq i \leq ep]$ the pattern P is a prefix of $\mathrm{T}[\mathrm{SA}[i], n]$, that is, P occurs at the positions $\mathrm{SA}[i]$ in T. It takes $O(m \log n)$ time to find the interval. Fig. 2 shows an example of a suffix array.

2.3. Generalized suffix trees

Each substring $\mathcal{C}[i, n]$, with $i \in [1, n]$, is called a *suffix* of $\mathcal{C}$. The *generalized suffix tree* (GST) of $\mathcal{C}$ is a path-compressed trie (i.e., unary paths are collapsed) in which all the suffixes of $\mathcal{C}$ are inserted. Internal nodes correspond to repeated strings of $\mathcal{C}$ and the

i	=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	=	A	T	A	#	T	A	A	A	#	T	A	T	A	#	$
SA	=	15	14	4	9	13	3	8	7	6	11	1	12	2	5	10

Fig. 2. Suffix array (SA) example for the example sequence $\mathcal{C}$ = ATA#TAAA#TATA#$. The positions $\mathrm{SA}(12, 15) = \langle 12, 2, 5, 10\rangle$ are those where the string "TA" appears.

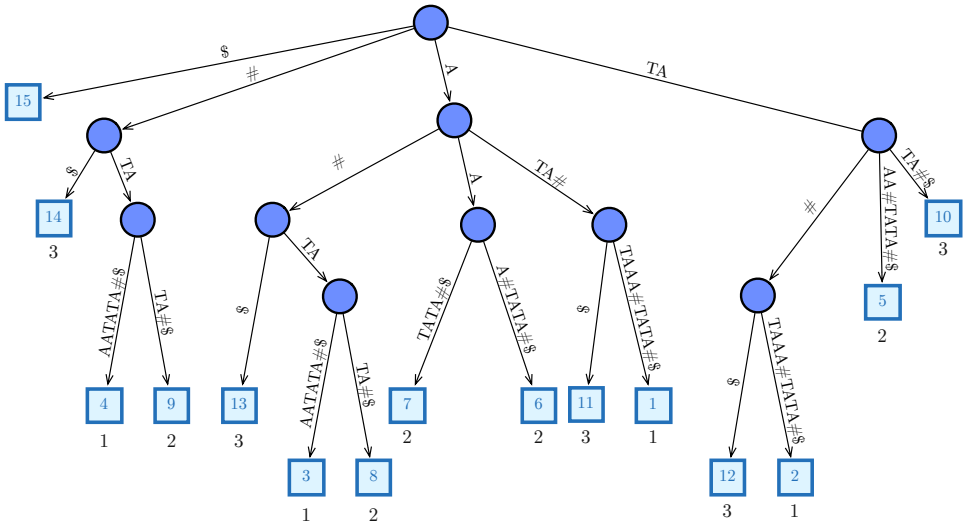


Fig. 3. Generalized suffix tree for the example sequence $\mathcal{C}$ = ATA#TAAA#TATA#$. The leaves point to the positions in $\mathcal{C}$ where the suffixes appear. The corresponding document numbers are written below.

leaves correspond to suffixes. For internal nodes v, $path(v)$ is the concatenation of the edge labels from the root to v. The suffix tree can list the occ occurrences of a pattern P of length m in optimal $O(m + occ)$ in time, by traversing from the root to the *locus* of P, i.e., the highest node v such that P is a prefix of $path(v)$. Then all the occurrences of P correspond to the leaves of the subtree rooted at v. These leaves correspond to the range $\mathrm{SA}[sp, ep]$. Indeed, v is the lowest common ancestor of the sp-th and the ep-th leaves. The suffix tree has $O(n)$ nodes and it uses $O(n)$ words of space. Fig. 3 shows an example of a generalized suffix tree built over the collection of Fig. 1.

We call $\mathrm{tf}(v, d)$ the number of leaves associated with document d that descend from the node v, that is, the number of times $path(v)$ appears in document d. This is a basic score measure (called "term frequency", hence the name tf) of the relevance of d for the pattern $path(v)$.

2.4. Compressed suffix arrays and self-indexes

The *Compressed Suffix Array* (CSA) (see Navarro and Mäkinen [2007]) can represent the text *and* its suffix array SA within the space of the compressed text, at most $n \log \sigma$ bits. This representation allows us to perform the following operations:

— SEARCH(P): determine the interval $\mathrm{SA}[sp, ep]$ corresponding to a pattern P.
— COUNT(P): return the number of occurrences of P (just $ep - sp + 1$).
— LOCATE(i): compute $\mathrm{SA}[i]$ for any i.

— EXTRACT(l, r): rebuild any $\mathrm{T}[l, r]$ from the original text.

Since the CSA is able to extract the original text from the index, it is considered as a replacement of T, and thus is called a *self-index*. A class of CSAs based on the so-called Ψ-function can compute $\mathrm{SA}[i]$ in $t_{\mathrm{l}} = O(\log n)$ time and are able to search for the pattern and count its occurrences in time $t_{\mathrm{s}}(m) = O(m \log n)$. Another class, called ***FM-Indexes*** [Ferragina and Manzini 2005], compute $\mathrm{SA}[i]$ in time $t_{\mathrm{l}} = O(\log^{1+\epsilon} n)$ for any $\epsilon > 0$ and provides search for P in time $t_{\mathrm{s}}(m) = O(m \log \sigma)$. Those are practical complexities achieved in implemented CSAs; better complexities are possible in theory. We refer the reader to surveys or books for more details [Navarro and Mäkinen 2007; Navarro 2016].

2.5. Bitmaps

Given a binary sequence $\mathrm{B}[1, n]$ one can build a data structure [Clark 1996; Munro 1996] that requires $n + o(n)$ bits and supports in $O(1)$ time the following operations:

— $\mathrm{RANK}_b(\mathrm{B}, i)$: number of occurrences of bit b in $\mathrm{B}[1, i]$.
— $\mathrm{SELECT}_b(\mathrm{B}, j)$: position in B of the jth occurrence of bit b.
— ACCESS(B, k) : the kth bit from bitmap B, i.e., $\mathrm{B}[k]$.

In practice these operations require a space overhead, on top of the n bits, of around $0.05n$–$0.25n$ bits, and the operations run in less than a microsecond [Vigna 2008; Gog and Petri 2014].

Raman et al. [2007] introduced another constant-time solution that compresses the bitmaps that have few 0s or few 1s. Okanohara and Sadakane [2007] presented a data structure called SDArray that performs better on very sparse bitmaps (very few 1s, $m \ll n$). It can support SELECT_b in constant time, and RANK_b and ACCESS in time $O(\log(n/m))$. Recently, Karkkainen et al. [2014] introduced the *hybrid bit vector*, which divides the bitmap into blocks and then chooses the encoding of each block separately from the techniques previously described.

2.6. Compact ordinal trees

There are $\Theta(4^n/n^{3/2})$ general trees containing n nodes, therefore one needs $\log_2(4^n/n^{3/2}) = 2n - \Theta(\log n)$ bits to represent any tree. Many compact representations that implement the operations in $O(1)$ time while requiring $2n + o(n)$ bits have been proposed [Munro and Raman 2002; Benoit et al. 2005; Navarro and Sadakane 2014]. A practical comparison [Arroyuelo et al. 2010] showed that, in practice, the best structures use 2.1–2.3 bits per node and solve the operations within microseconds. The representation of Navarro and Sadakane [2014] was shown to be the fastest in practice among those supporting full functionality. In this paper we use it to solve queries PREORDER(v) (the preorder of node v), PREORDER_SELECT(i) (the ith node in preorder), DEPTH(v) (depth of node v), SUBTREE_SIZE(v) (number of nodes in subtree rooted at v), LEAF_RANK(v) (number of leaves to the left of v), LEAF_SELECT(i) (the ith leaf left to right), and LCA(u, v) (lowest common ancestor of nodes u and v). This representation uses about $2.3n$ bits in practice.

2.7. Range maximum queries

The *Range Maximum Query* (RMQ) over an array A is defined as $\mathrm{RMQ}_A(i, j) = \mathrm{argmax}_{i \le k \le j} A[k]$. It is possible to solve this query in constant time after preprocessing A and storing a structure using $2n + o(n)$ bits that does not access A at query time [Fischer and Heun 2011]. This data structure can be configured to answer range minimum queries as well. In practice its time is close to that of ordinal tree operations.

2.8. Direct access codes

A recurrent problem of variable-length representations of integer sequences is how to directly access the ith number in the sequence. Brisaboa et al. [2013] presented a simple and practical solution called *Direct Access Codes*. The idea is to pack each variable-length integer into chunks of length b. Then the chunks are rearranged to allow the access of any ℓ-bit number in the sequence in time $O(\ell/b)$. The maximum space overhead for a number of ℓ bits is $\ell/b + b$ bits.

2.9. Wavelet trees

The *wavelet tree* [Grossi et al. 2003] of an integer sequence $\mathcal{S}[1, n]$ containing σ different symbols, is a binary balanced tree that stores a bitmap in every node except the leaves. Every position of the root bitmap is a '0' or '1' depending whether the symbol at the corresponding position of $\mathcal{S}$ belongs to the first or the second half of the alphabet. The symbols that are marked with a '0' are recursively represented in the left subtree of the root, while those marked with a '1' are representing on the right subtree. The halving of the alphabet continues recursively until the leaves, which represent single symbols. The wavelet tree has σ leaves, each level of the tree requires n bits, and the height of the tree is $\log \sigma$. Therefore, the wavelet tree needs $n\lceil \log \sigma \rceil$ bits, just like a plain representation of $\mathcal{S}$.

The wavelet tree can retrieve any symbol $\mathcal{S}[i]$ in time $O(\log \sigma)$ if ACCESS and RANK_b operations are enabled on its bitmaps, and therefore it can act as a replacement of $\mathcal{S}$ (this increases the space in $o(n \log \sigma)$ bits). To recover $\mathcal{S}[i]$, it checks if the root bitmap has a 0 at position i. In this case, the symbol belongs to the left part of the alphabet, so it continues on the left with the new position $i = \text{RANK}_0(i)$. Otherwise, it continues on the right with the position $i = \text{RANK}_1(i)$. When it reaches a leaf, its symbol is $\mathcal{S}[i]$. With a similar procedure we can support query $\text{RANK}_c(\mathcal{S}, i)$, which counts the number of occurrences of symbol c in $\mathcal{S}[1, i]$.

Wavelet trees can also be used to represent an $n \times r$ grid that contains n points, one per column [Mäkinen and Navarro 2006]. The root represents the sequence of coordinates y_i of the points in x-coordinate order. It only stores a bitmap $B[1, n]$ telling at $B[i]$ whether $y_i < r/2$. Then the points with $y_i < r/2$ are represented, recursively, on the left child of the root, and the others on the right. Adding RANK_b capabilities to the bitmaps, the wavelet tree requires overall $n \log r(1 + o(1))$ bits and can track any point towards its leaf (where the y_i value is revealed) in time $O(\log r)$, just as we have shown to retrieve $\mathcal{S}[i]$. It can also count, in $O(\log r)$ time, the number of points lying inside a rectangle $[x_1, x_2] \times [y_1, y_2]$: Start at the root with the interval $[x_1, x_2]$ and project those values towards the left and right children (on the left child the interval is $[\text{RANK}_0(B, x_1 - 1) + 1, \text{RANK}_0(B, x_2)]$, and similarly with RANK_1 on the right). This is continued until reaching the $O(\log r)$ wavelet tree nodes that cover $[y_1, y_2]$. Then the answer is the sum of the lengths of the mapped intervals. One can also track those points toward the leaves and report them, each in time $O(\log r)$. Fig. 4 shows the wavelet tree of an example sequence $\mathcal{S}$ and a grid interpreting $\mathcal{S}$ as a sequence of values y_i.

2.10. Wavelet trees and RMQs

Navarro et al. [2013] introduced compact data structures for various queries on two-dimensional weighted points, including range top-k queries. They enhance the bitmap of each node as follows: Let $x_1, \ldots, x_s$ be the points represented at a node, and $w(x)$ be the weight of point x. Then a range maximum query (RMQ) data structure built on $w(x_1), \ldots, w(x_s)$ is stored together with the bitmap. Such a structure uses $2s + o(s)$ bits and finds the maximum weight in any range $[w(x_i), \ldots, w(x_j)]$ in constant time,

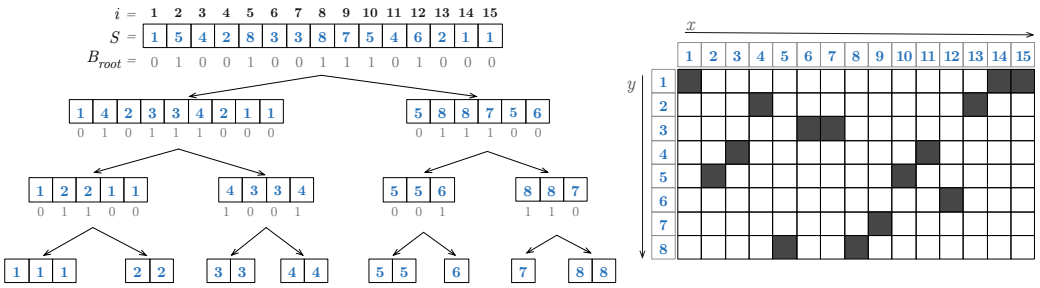


Fig. 4. On the left, the wavelet tree of the sequence $S = 154283387546211$. The bitmaps are represented below the sequences in grey. For example, B_{root} represents the bitmap corresponding to the root of the wavelet tree. On the right, the grid interpretation of the same sequence.

as we have seen, without accessing the weights themselves. Therefore, the total space becomes $3n \lg r + o(n \log r)$ bits. To solve top-k queries on a grid range $Q = [x_1, x_2] \times [y_1, y_2]$, we first traverse the wavelet tree to identify the $O(\log r)$ bitmap intervals where the points in Q lie (a counting query would, at this point, just add up all the bitmap interval lengths). The heaviest point in Q in each bitmap interval is obtained with an RMQ, but we need to obtain the actual priorities in order to find the heaviest among the $O(\log r)$ candidates. The weights are stored sorted by y-coordinate, so we obtain each one in $O(\log r)$ time by tracking the point with maximum weight in each interval. Thus a top-1 query is solved in $O(\log^2 r)$ time. For a top-k query we must maintain a priority queue of the candidate intervals, and each time the next heaviest element is found, we remove it from its interval and reinsert in the queue the two resulting subintervals. The total query time is $O((k + \log r) \log(kr))$.

It is possible to reduce the time to $O((k+\log m) \log^\epsilon m)$ time and $O(\frac{1}{\epsilon} n \log m)$ bits, for any constant $\epsilon > 0$ [Navarro and Nekrich 2012], but the space usage grows fast.

2.11. K^2-trees

The K^2-tree [Brisaboa et al. 2014] is a data structure to compactly represent sparse binary matrices (which can also be regarded as point grids). The K^2-tree subdivides the matrix into K^2 submatrices of equal size. The submatrices are considered in row-major order, top-to-bottom and left-to-right, and each is represented with a bit, set to 1 if the submatrix contains at least one non-zero cell. Each node whose bit is 1 is recursively decomposed, subdividing its submatrix into K^2 children, and so on. The subdivision ends when a fully-zero submatrix is found or when we reach the individual cells. All these bits are concatenated levelwise into a bitvector T, which is enhanced with RANK_b functionality to allow efficient traversals.

The K^2-tree can answer range queries efficiently in practice, although it has no good worst-case time guarantees. The worst-case space, if t points are in an $n \times n$ matrix, is $K^2\, t \log_{K^2} \frac{n^2}{t}(1 + o(1))$ bits. This can be reduced to $t \log \frac{Kn^2}{t}(1 + o(1))$ bits if the bitmaps are compressed. This is similar to the wavelet tree space, but in practice K^2-trees use much less space when the points are clustered. Fig. 5 shows an example.

2.12. K^2-treaps

Brisaboa et al. [2016] introduced a modified version of the K^2-tree that is able to efficiently report the top-k points that lie inside a rectangle $Q = [x_1, x_2] \times [y_1, y_2]$ of an $n \times n$ grid. Consider the case where the cells of a matrix $M[n \times n]$ contain either an empty value or a weight in the range $[1, w]$. The K^2-treap is then constructed by performing a quadtree-like recursive partition of M into K^2 submatrices just as the K^2-tree.

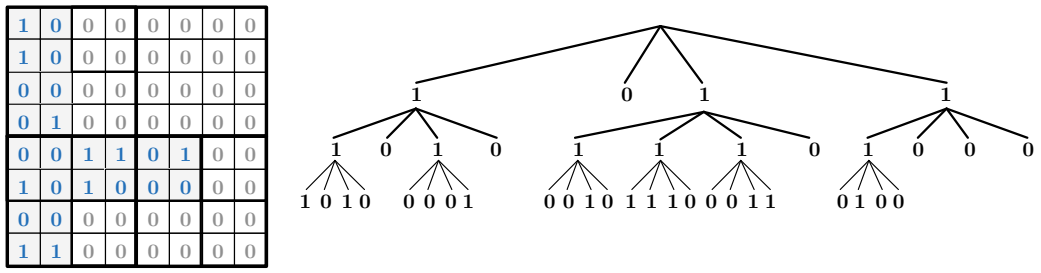


Fig. 5. On the left, a sparse binary matrix. On the right, its corresponding K^2-tree with $K = 2$. The actual representation is the bitvector $T = 10111010111010001010000100101111000110100$ obtained by reading the tree bits levelwise.

Now, at each step of the partitioning, the K^2-treap stores the coordinate of the cell with the maximum weight of the partition and the corresponding weight. That value is deleted from the matrix and the subdivision continues until a fully-zero submatrix is found or when an individual cell is reached. To solve top-k queries on a grid range $Q = [x_1, x_2] \times [y_1, y_2]$, the process initializes a max-priority queue containing the root node of the K^2-treap. Now, we iteratively pop the priority queue. If the maximum-weight point stored with the extracted node lies inside Q, we report it as the next answer. In either case we push all the children from the extracted node whose submatrix intersects with Q and iterate. The process finishes when k results have been reported or when the priority queue is empty.

2.13. Muthukrishnan's algorithm

Muthukrishnan [2002] introduced the first solution for retrieving the documents from a collection that contain a pattern P, in optimal time and linear space. The idea is based on suffix arrays and introduces a new data structure called *document array* $\mathrm{DA}[1, n]$. The algorithm starts by constructing the suffix array $\mathrm{SA}[1, n]$ from the concatenation $\mathcal{C}$ of the collection of documents $\mathcal{D}$, and for every position pointed from $\mathrm{SA}[i]$ it stores in $\mathrm{DA}[i]$ the document number where the suffix belongs. The document array requires $n \log D$ bits. In order to list all the distinct documents from the range $\mathrm{SA}[sp, ep]$ of the occurrences of P, Muthukrishnan lists all the distinct values in $\mathrm{DA}[sp, ep]$. The idea is to use another array $\mathrm{CA}[1, n]$ where $\mathrm{CA}[i] = \max\{j < i,\ \mathrm{DA}[j] = \mathrm{DA}[i]\} \cup \{-1\}$, which is preprocessed for range minimum queries. Each value $\mathrm{CA}[m] < sp$ for $sp \leq m \leq ep$ corresponds to a distinct value $\mathrm{DA}[m]$ in $\mathrm{DA}[sp, ep]$. A range minimum query in $\mathrm{CA}[sp, ep]$ gives one such value m, and we continue recursively on the intervals $[sp, m-1]$ and $[m+1, ep]$ until the minimum is $\geq sp$. This way, it is possible to retrieve any amount k of unique elements from DA in time $O(k)$. Fig. 6 shows the document array (DA) and array CA for the example collection $\mathcal{C}$.

Algorithm 1 shows the procedure to list all the distinct elements of $\mathrm{DA}[sp, ep]$, extended to the case where we already have a set of documents d and want to complete it up to size k (this is the variant we will use in this article). In the initial call we set $gsp = sp$.

3. RELATED WORK

In this section we cover the state-of-the-art of solutions for top-k document retrieval. As there exists a recent and thorough survey [Navarro 2014] and our focus is practical, we will only describe the practical implementations. We also leave aside some proposals [Belazzougui et al. 2013; Ferrada and Navarro 2014; Navarro et al. 2014a] that are useful on particular cases, but not competitive for the mainstream problem.

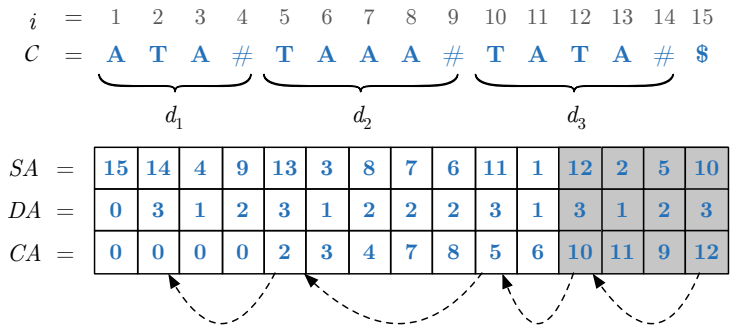


Fig. 6. Document array (DA) and array CA for the sequence shown in Fig. 1. The positions $\mathrm{SA}(12, 15) = \langle 12, 2, 5, 10\rangle$ are those where the string "TA" appears. The dotted arrows represent a list of pointers to the next smaller position in CA where d_3 occurs.

ALGORITHM 1: Algorithm for completing set d with up to k distinct documents from $\mathrm{DA}[sp, ep]$.

Input: start position sp, end position ep, current document set d, document array $\mathrm{DA}[1, n]$, RMQ data structure over array CA, the size s to which d should grow, and a copy of the original starting position gsp.
Output: set of distinct documents d in $\mathrm{DA}[sp, ep]$ enlarged to size s.

```
00 DOCUMENT_LISTING(sp, ep, d, k, gsp)
01   if sp > ep or |d| ≥ k then return d
02   p ← RMQ(sp, ep)
03   if CA[p] < gsp then
04     doc ← DA[p]
05     if doc ∉ d then d ← d ∪ {doc}
06     d ← d ∪ DOCUMENT_LISTING(sp, p − 1, d, k, gsp)
07     d ← d ∪ DOCUMENT_LISTING(p + 1, ep, d, k, gsp)
08   return d
```

The practical solutions build on the CSA of $\mathcal{C}$ and its document array DA. Once we establish that the interval of P is $\mathrm{SA}[sp, ep]$, the query is solved by finding the k values that appear most often in $\mathrm{DA}[sp, ep]$.

GREEDY. Culpepper et al. [2010] studied various heuristics to solve top-k queries on top of a CSA and a wavelet tree of the document array DA, based on the ability of the wavelet tree of retrieving all the distinct documents of $\mathrm{DA}[sp, ep]$ and their frequencies [Gagie et al. 2012]. We describe here their best variant, called *greedy traversal*. Their technique relies on the fact if P appears many times in a document, then the $[sp, ep]$ interval on the path to that document leaf in the wavelet tree should also be long. The wavelet tree is traversed in a greedy fashion, selecting the longest $[sp, ep]$ interval first. Then the algorithm will first reach the leaf with the highest tf value. The next document reported will be the second-highest score, and so on. This procedure continues until k documents are reported.

Culpepper et al. [2012] adapted the scheme to large natural language text collections (where each word is taken as an atomic symbol), showing that it was competitive with inverted indexes for some queries (see previous work on this line by Patil

et al. [2011]). The solution, however, resorts to approximations when handling ranked Boolean queries.

NPV. Apart from their linear-space index, Hon et al. [2009] proposed a succinct index based on sampling suffix tree nodes and storing top-k answers on those. When the locus of P reaches a non-sampled node, they choose the highest sampled node below the locus and *correct* its precomputed answer by considering the additional leaves of the locus. The sampling ensures that this work is bounded.

Navarro et al. [2014b] implemented a practical version of this proposal, and also combined it with the greedy algorithm of Culpepper et al. [2010] to speed up the correction process. They also studied compressed representations of the wavelet tree of DA, by using grammar compression (specifically, RePair [Larsson and Moffat 2000]) on its bitmaps. They carried out a thorough experimental study of these approaches and previous ones, establishing one of the first baselines for future comparisons of top-k indexes.

SORT. The first successful attempt to engineer an index handling hundred-gigabyte collections, having millions of documents and containing large alphabets, was presented by Gog et al. [2014]. They build a framework for experimentation with succinct data structures, where they reimplement the greedy approach of Culpepper et al. [2010] in this scenario.

Gog et al. [2014] also implement a basic folklore solution to the top-k problem, called SORT. This is based on simply collecting all the values in $\mathrm{DA}[sp, ep]$, sorting them by document identifier, computing term frequencies, and choosing the k largest ones. This is a good baseline to evaluate if more sophisticated ideas are worthy.

Patil et al. One of the first implementations of practical top-k document retrieval was introduced by Patil et al. [2011]. In their work, they store for some nodes the complete inverted lists containing the document identifiers and the frequency of the string represented by the node. These inverted lists are then sorted by pre-order rank of the strings in the GST and stored contiguously in an array. For a given pattern P they find the locus in the GST and then map the preorder rank of the locus and its rightmost leaf. This creates a range in the array of inverted lists, which is found by performing a binary search, and then they are able to find the top-k documents using RMQ queries over the frequencies of the inverted lists that lie within that range. In practice, this solution requires 5 to 19 times the size of the collection, which makes it unpractical for most scenarios. For this reason, we do not compare our work to this solution.

State of the art. Current implementations need about 2–5 times the size of the collection and answer queries in tens of milliseconds. The simple implementation of the solution of Hon et al. [2009] requires 2–5 times the size of the collection and answers queries in about 10–100 milliseconds. The greedy approach introduced by Culpepper et al. [2010] improves the space to 2–4 times the size of the collection and reduces the time to 1–20 milliseconds. The best results obtained by Navarro et al. [2014b] use 2–3 times the size of the collection and answer queries in about 1–10 milliseconds. These implementations are limited to relatively small collections of a few hundred megabytes, because of construction issues or for using 32-bit offsets. Moreover, implementations are tailored to sequences over small alphabets ($\sigma \leq 255$). Table I shows a summary of these results, compared to our achievements in this article.

Table I. Comparison of practical results.

Index	Max. σ	Max. $\|\mathcal{C}\|$ in MB	Size (times $\|\mathcal{C}\|$)	Time in μs
Hon et al. implementation [Navarro et al. 2014b]	2^8-1	137	$2-5$	10^4-10^5
GREEDY [Culpepper et al. 2010]	2^8-1	100	$2-4$	10^3-10^4
NPV [Navarro et al. 2014b]	2^8-1	137	$2-3$	10^3-10^4
[Patil et al. 2011]	2^8-1	100	$5-19$	10^2-10^3
SORT [Gog et al. 2014]	$2^{64}-1$	72,000	$2-3$	10^4-10^6
Ours	$2^{64}-1$	72,000	$2-3$	10^1-10^3

4. THE OPTIMAL-TIME SOLUTION

This section describes in detail the top-k framework of Hon et al. [2009] and the subsequent optimal-time solution of Navarro and Nekrich [2012], which is the one we implement in this article.

4.1. Hon et al. solution

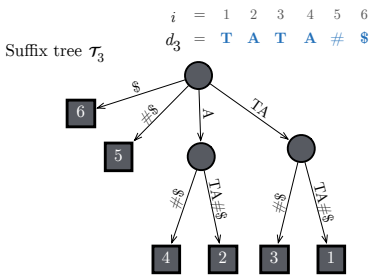
Let $\mathcal{T}$ be the suffix tree of the concatenation $\mathcal{C}$ of a collection of documents $d_1, d_1, \ldots, d_D$. This tree contains the nodes corresponding to all the suffix trees $\mathcal{T}_i$ of the documents d_i: for each node $u \in \mathcal{T}_i$, there is a node $v \in \mathcal{T}$ such that $path(v) = path(u)$. We will say that $v = map(u, i)$. Also, let $parent(u)$ be the parent of u and $depth(u)$ be its depth. The idea of Hon et al. [2009] is to store $\mathcal{T}$ with additional information about the trees $\mathcal{T}_i$. For each $v = map(u, i)$, they store a pointer $ptr(v, i) = v' = map(parent(u), i)$, noting where the parent of u maps in $\mathcal{T}$. These pointers are stored at the target nodes v' in a so-called F-list. Together with the pointers $ptr(v, i)$ they also store a weight $w(v, i)$, which is the relevance of $path(u)$ in d_i. This relevance can be any function that depends on the set of starting positions of $path(u)$ in d_i. In this paper we focus on a simple one: the number of leaves of u in $\mathcal{T}_i$, that is, $\mathrm{tf}(P, d)$. Let v be the locus in $\mathcal{T}$ of a pattern P. They proved that, for each distinct document d_i where P appears, there is exactly one pointer $ptr(v'', i) = v'$ going from a descendant v'' of v (v itself included) to a (strict) ancestor v' of v, and $w(v'', i)$ is the relevance of P in d_i. Obtaining the top-k documents using this structure boils down to identifying all the pointers in the F-lists of the ancestors of v that come from the subtree of v, and then selecting k corresponding documents with the highest scores. This task can be carried out in $O(m + k \log k)$ time and the index requires linear space.

Figure 7 shows this structure on our example collection $\mathcal{C}$. The top left part of the figure shows the the suffix tree $\mathcal{T}_3$ corresponding to document d_3. The bottom part shows the generalized suffix tree $\mathcal{T}$ of the collection $\mathcal{C}$. The numbers inside the nodes correspond to the preorder number and the dark nodes are those mapped from $\mathcal{T}_3$ to $\mathcal{T}$. The dotted lines correspond to the pointers $ptr(v, i) = map(parent(u), i)$.

We store the documents and weights as pairs $(d_i, w(v, i))$ and append them to the F-lists, which are associated with the node where the pointer arrives. These lists are shown on the top right part of the figure.

4.2. Optimal solution

Navarro and Nekrich [2012] presented an optimal $O(m+k)$ time solution that requires linear space, $O(n \log n)$ bits, for solving the top-k document retrieval problem. They represent the structure of Hon et al. [2009] as a grid of size $O(n) \times O(n)$ with labeled weighted points. Each pointer $v' = ptr(v, i)$ is represented as a point in the grid, whose x-coordinate is the preorder number of the source v, and whose y-coordinate is the depth of the target v'. Then, the pointers going from the subtree of v to an ancestor of v correspond to the points whose x-coordinate is the preorder range of the subtree of v,



Node	F-Lists (d,w)
v_1	(3,1)(1,1)(2,1)(3,2)(1,2)(2,3)(3,2)(1,1)(2,1)
v_4	(3,1)(1,1)(2,1)(2,2)(3,1)(1,1)
v_7	(2,1)(2,1)
v_9	(3,1)(1,1)(2,1)(3,1)

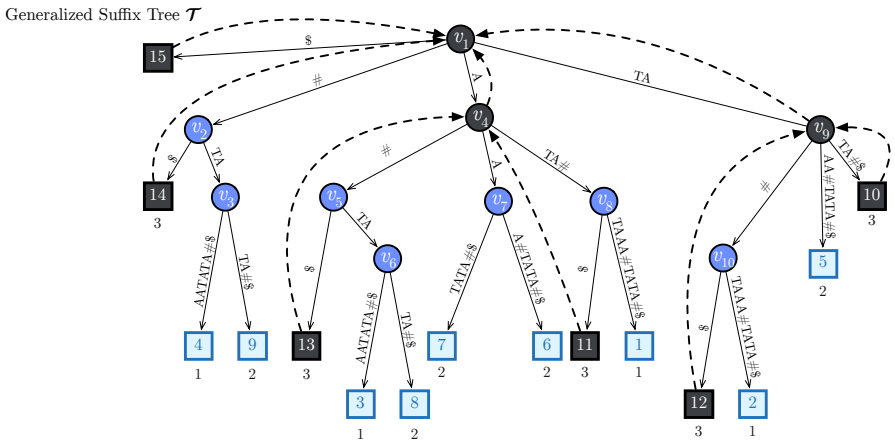


Fig. 7. Top left: The suffix tree $\mathcal{T}_3$ of document d_3. Bottom: The generalized suffix tree of $\mathcal{T}$ with preorder naming of the inner nodes and the corresponding document numbers below the leaves. The dark nodes represent the nodes mapped from $\mathcal{T}_3$ to $\mathcal{T}$. The dotted lines corresponds to the pointers $ptr(v,i) = map(parent(u),i)$. Top right: the nonempty F-lists generated containing elements as pairs $(d_i, w(v,i))$.

and whose y-coordinates are smaller than the depth of v. By giving weights $w(v,i)$ to the points, the problem boils down to retrieving the k heaviest points in that 3-sided query range.

The construction procedure is the following: they traverse $\mathcal{T}$ in preorder, where they add for each node $v \in \mathcal{T}$ and pointer $ptr(v,i) = v'$, a new rightmost x-coordinate with only one point, with a y-coordinate value set to $depth(v')$, weight equal to $w(v,i)$ and label equal to i. To solve a query, they find the locus v of P, determine the range $[x_1, x_2]$ of all the x-coordinates filled by v or its descendants, find the k heaviest points in $[x_1, x_2] \times [0, depth(v) - 1]$, and report their labels. A linear-space representation allows them to carry out this task in optimal time. Figure 8 shows an example of this procedure: the top part shows the generalized suffix tree $\mathcal{T}$. The grey area corresponds to the locus of the pattern "TA" and the corresponding subtree. The bottom part of the figure shows a two-dimensional grid that maps the content of the F-lists. The grey area shows the mapping from the query pattern to a range $[x_1, x_2]$ in this grid corresponding to the subtree of the locus. The dark grey area is the final range $[x_1, x_2] \times [0, depth(v) - 1]$ from where the procedure finally selects the top-k highest-weighted points.

5. A BASIC COMPACT IMPLEMENTATION

In this section we describe a basic compact implementation of the optimal-time solution [Navarro and Nekrich 2012], which will be labeled WTRMQ.

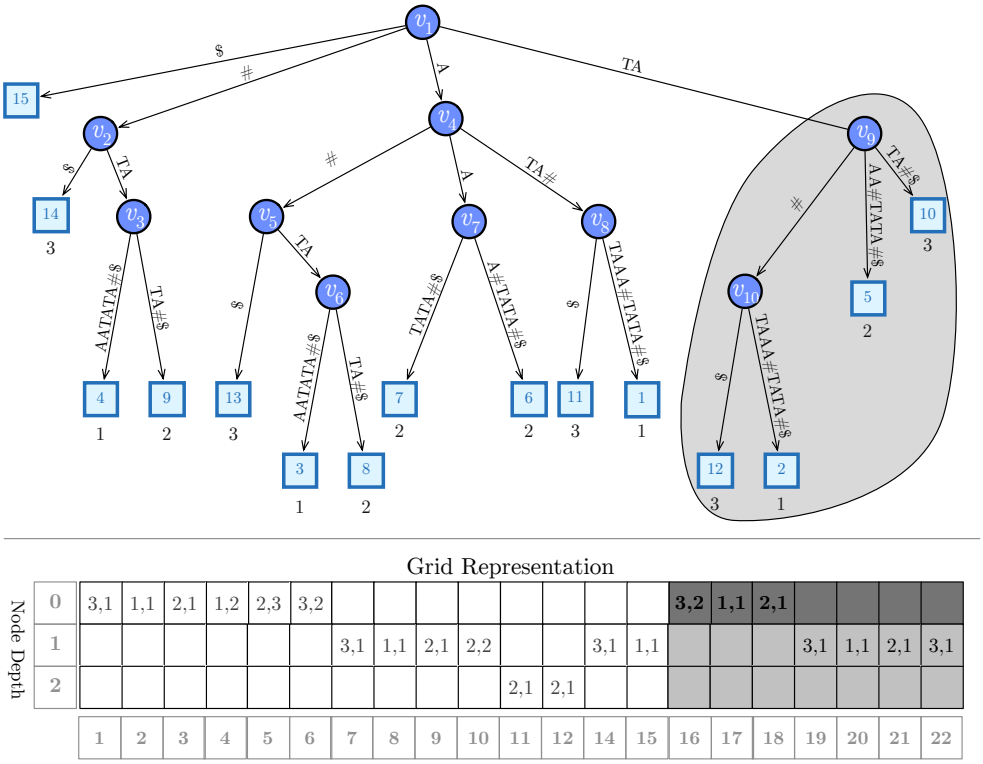


Fig. 8. Top: Suffix tree of $\mathcal{C}$ with preorder naming of the inner nodes. The subtree of v_9, the locus of pattern "TA", is highlighted in grey. Bottom: Navarro and Nekrich [2012] geometric representation of the F-lists of Hon et al. [2009]. Each cell contains a pair $(d_i, w(v,i))$. The grey area in the grid represents all the pairs that are generated from pointers that belong to the tree rooted at v_9. The darkest grey area represents the final range $[x_1, x_2] \times [0, depth(v) - 1]$.

5.1. Searching for patterns

In order to search for patterns in the collection, we use a CSA, which computes the interval $[sp, ep]$ corresponding to P in time $O(t_s(m))$ for some function $t_s(m)$, where $m = |P|$. We also add a compact representation of the topology of the suffix tree, seeing it as an ordinal tree. Note that this is just the topology, not a full suffix tree, so we need to search using the CSA. We will use the suffix tree topology to obtain the locus v of P from sp and ep, and then to map it to the grid.

5.2. Mapping to the grid

The grid G is of width $\sum_i |\mathcal{T}_i| \leq 2n$, as we add one coordinate per node in the suffix tree of each document. To save space, we will consider a *virtual* grid just as defined, but will store a narrower *physical* grid. In the physical grid, the entries corresponding to leaves of $\mathcal{T}$ (which contain exactly one pointer $ptr(v,i)$) will not be represented. Thus the physical grid is of width at most n. This *frequency thresholding* is a key idea, as it halves the space of most structures in our index.

A bitmap $B[1, 2n]$ will be used to map between the suffix tree and the grid. Bitmap B will mark starting positions of nodes of $\mathcal{T}$ in the physical grid: along the grid construction process, each time we arrive at an internal node v we add a 1 to B, and each

time we add a new x-coordinate to the grid (due to a pointer $ptr(v, i)$) we add a 0 to B. This structure will be called MAP.

5.3. Finding isolated documents

Due to frequency thresholding, the information on the grid may be insufficient to answer top-k queries when the result includes documents where P appears only once. For this purpose, we store the RMQ structure on top of the array CA of Muthukrishnan [2002] (the array itself is not stored). This structure, RMQC, allows us list k distinct documents in any interval $\mathrm{SA}[sp, ep]$.

To determine the identity of a document found at position i of CA, we will use the CSA to compute $p = \mathrm{SA}[i]$ and then determine to which document position $\mathcal{C}[p]$ belongs. For this sake we store a sparse bitmap DOCBITMAP, that marks beginnings of documents in $\mathcal{C}$. The document is then $\textsc{rank}_1(\mathrm{DOCBITMAP}, p)$.

5.4. Representing the grid

In the grid there is exactly one point per x-coordinate. We represent with a wavelet tree [Grossi et al. 2003] the sequence of corresponding y-coordinates. Each node v of the wavelet tree represents a subsequence of the original sequence of y-coordinates. We consider the (virtual) sequence of the weights associated to the points represented by v, $W(v)$, and build an RMQ data structure for each node $W(v)$, as explained, to support top-k queries on the grid.

5.5. Representing labels and weights

The labels of the points, that is, the document identifiers, are represented directly as a sequence of at most $n \log D$ bits, aligned to the bottom of the wavelet tree. Given any point to report, we descend to the leaf and retrieve the aligned document identifier.

The weights are stored similarly, but using direct access codes, to take advantage of the fact that most weights (term frequencies) are small.

5.6. Summing up

The WTRMQ index consists of eight components: the compressed suffix array (CSA), the suffix tree topology (TREE), the suffix tree to grid mapping (MAP), the wavelet tree over the grid G (WT) including the RMQ structures at each node, the document ids associated with the grid elements (DOC), in the same order of leaves of the wavelet tree, the weights associated to the documents (FREQ), the RMQ structure to retrieve documents occurring once over the CA array (RMQC), and the bitmap marking document borders (DOCBITMAP).

If the height of the suffix tree is $O(\log n)$, as is the case in many reasonable distributions [Szpankowski 1993], then WT requires $3n \log\log n + O(n)$ bits. The other main components are $|\mathrm{CSA}| \leq n \log \sigma$ bits, the array DOC, which uses $n \log D$ bits, and the array FREQ, which in the worst case needs $n \log n$ bits but uses much less in practice. The other elements add up to $O(n)$ bits, with a constant factor of about 8–10.

5.7. Answering queries

The first step to answer a query is to use the CSA to determine the range $[sp, ep]$. To find the locus v of P in the topology of the suffix tree, we compute l and r, the spth and epth leaves of the tree, respectively, using $l = \textsc{leaf\_select}(sp)$ and $r = \textsc{leaf\_select}(ep)$; then we have $v = \textsc{lca}(l, r)$.

To determine the horizontal extent $[x_1, x_2]$ of the grid that corresponds to the locus node v, we first compute $p_1 = \textsc{preorder}(v)$ and $p_2 = p_1 + \textsc{subtree\_size}(v)$. This gives the preorder range $[p_1, p_2)$ including leaves. Now $l_1 = \textsc{leaf\_rank}(p_1)$ and $l_2 = \textsc{leaf\_rank}(p_2 - 1)$ give the number of leaves up to those preorders. Then, since we have

omitted the leaves in the physical grid, we have $x_1 = \text{SELECT}_1(B, p_1 - l_1) - (p_1 - l_1) + 1$ and $x_2 = \text{SELECT}_1(B, p_2 - l_2) - (p_2 - l_2)$. The limits in the y axis are just $[0, \text{DEPTH}(v) - 1]$. Thus the grid area to query is determined with a constant amount of operations on bitmaps and trees.

Once the range $[x_1, x_2] \times [y_1, y_2]$ to query is determined, we proceed to the grid. We determine the wavelet tree nodes that cover the interval $[y_1, y_2]$, and map the interval $[x_1, x_2]$ to all of them.

We now use the top-k algorithm for wavelet trees we described. Let $v_1, v_2, \ldots, v_s$ be the wavelet tree nodes that cover $[y_1, y_2]$ and let $[x_1^i, x_2^i]$ be the interval $[x_1, x_2]$ mapped to v_i. For each of them we compute $\text{RMQ}_{W(v_i)}(x_1^i, x_2^i)$ to find the position x_i with the largest weight among the points in v_i, and find out that weight and the corresponding document, w_i and d_i. We set up a max-priority queue that will hold at most k elements (elements smaller than the kth are discarded by the queue). We initially insert the tuples $(v_i, x_1^i, x_2^i, x_i, w_i, d_i)$, being w_i the sort key. Now we iteratively extract the tuple with the largest weight, say $(v_j, x_1^j, x_2^j, x_j, w_j, d_j)$. We report the document d_j with weight w_j, and create two new ranges in v_j: $[x_1^j, x_j - 1]$ and $[x_j + 1, x_2^j]$. We compute their RMQ, find the corresponding documents and weights, and reinsert them in the queue. After k steps, we have reported the top-k documents. We will refer to this procedure as TOPK_GRID_QUERY.

If we implement the bitmap, tree, and RMQ operations in constant time, and assuming again that the suffix tree is of height $O(\log n)$, the total time of this process is $O(t_s(m) + (k + \log\log n)(\log\log n + \log k))$. This is because there are $O(\log\log n)$ wavelet tree nodes covering $[y_1, y_2]$, and for each of them we descend to the leaf to find the weight and the document identifier (again in $O(\log\log n)$ time) and insert them in the priority queue ($O(\log k)$ time). Then we repeat k times the process of extracting a result from the queue, and generating two new weights to insert in it.

We remind the reader that we have not stored the leaves in the grid. Therefore, if the procedure above yields less than k results, we must complete it with documents where the pattern appears only once. Here we use Algorithm 1 to complete our current result d to make it of size k by adding new documents of $\text{DA}[sp, ep]$. Here we use RMQC and DOCBITMAP. We might have to extract up to k distinct documents with this technique to complete the answer (since we may obtain again those we already have from the grid). Each step requires $O(t_1)$ time to obtain the document identifier, where t_1 is the time needed by the CSA to return the content of a suffix array cell, so this may add up to time $O(k\, t_1)$ to the complexity.

Algorithm 2 shows the complete procedure.

6. AN IMPROVED INDEX

In this section we present the two main ideas that lead to a conceptually simpler and, as we will see later, faster and smaller representation than the basic one (WTRMQ).

6.1. Mapping the suffix tree to the grid

We reorganize the grid so that a more space-efficient mapping from pattern P to a 3-sided range query in the grid is possible. The basic solution uses $2t$ bits to represent the topology of the suffix tree (of t nodes, where $n \le t \le 2n$), plus up to $2n$ bits for B, for a total of up to $6n$ bits (plus the sublinear part). The improved representation will consist of a single bitvector H, of length $2(n - D)$, and the mapping from the suffix tree to the grid will be simpler.

We first explain how we list the parent pointers $ptr(v, i)$ leaving the suffix tree nodes v in the grid, and then show how we can efficiently map to the grid. We identify each internal node with the position in SA of the rightmost leaf of its leftmost subtree.

ALGORITHM 2: Top-k algorithm using WTRMQ index

Input: Query pattern P, amount k of documents to retrieve
Output: top-k documents

```
00  TOPK(P, k)
01    result ← ∅
02    [sp, ep] ←LOCATE(P)
03    l ← LEAF_SELECT(sp)
04    r ← LEAF_SELECT(ep)
05    v ← LCA(l, r)
06    p1 ← PREORDER_RANK(v)
07    p2 ← p+SUBTREE_SIZE(v)
08    l1 ←LEAF_RANK(p1)
09    l2 ←LEAF_RANK(p2 − 1)
10    x1 ← SELECT1(B, p1 − l1) + 1
11    x2 ← SELECT1(B, p2 − l2) − (p2 − l2)
12    depth ← DEPTH(v) − 1
13    result ← TOPK_GRID_QUERY(x1, x2, 0, depth, k)
14    if |result| < k then
15      result ← DOCUMENT_LISTING(sp, ep, result, k, sp)
16    return result
```

Fig. 9 shows this naming for the internal nodes: The root node is named v_1 because the position of its first child (which is also a leaf) is at position 1, the rightmost leaf of the leftmost child of node v_7 is at position 7, and so on. Note that the names are between 1 and n and they are unique (although not all names must exist, e.g., there is no node v_4 in our suffix tree). The bitvector H is generated by first writing n 1s and then inserting a 0 right before the j-th 1 per pointer $ptr(v_j, i)$ leaving node v_j.

The 0s in H then correspond to the x-domain in the grid, thus we do not need to represent it explicitly. On the other hand the y-coordinate (string depth of target node), weight and document id of the pointers, are stored associated to the corresponding 0 in H (in arrays y, FREQ, and DOC, respectively, see the right of Fig. 9). Now assume the CSA search yields the leaf interval $[sp, ep]$ for P. These are the positions of the leftmost and rightmost leaves that descend from the locus node v of P (although we will not compute v). Then, we note that a node v_j lies in the subtree of v (including v) if and only if $j \in [sp, ep - 1]$.

LEMMA 6.1. *A node v_j, if it exists, lies in the subtree of v (including v) if and only if $j \in [sp, ep - 1]$.*

PROOF. If v_j is in the subtree of v, then its range of descendant leaves is included in that of v, $[sp, ep]$. Since there are no unary nodes, the rightmost child v_r of v_j has at least one descendant leaf, and thus the leaves descending from the leftmost child v_l of v_j are within $[sp, ep - 1]$. In particular, the rightmost leaf descending from v_l, j, also belongs to $[sp, ep - 1]$. Conversely, if $j \in [sp, ep - 1]$ and v_j exists, then v_j is an ancestor of leaf j and so is v, thus they are the same or one descends from the other. However, if v descended from v_j, then j could only be ep (if p was the leftmost child v_l of v_j or belonged to the rightmost path descending from v_l), or it would be outside $[sp, ep]$. □

Therefore, all the pointers leaving from nodes in the subtree of v are stored contiguously in the grid, and can be obtained by finding the 0s that are between the $(sp - 1)$th and the $(ep-1)$th 1 in H (as the 0s are placed before their corresponding 1). Note that no LCA operation on any suffix tree topology is necessary, only operation $\text{SELECT}_1(H, p)$.

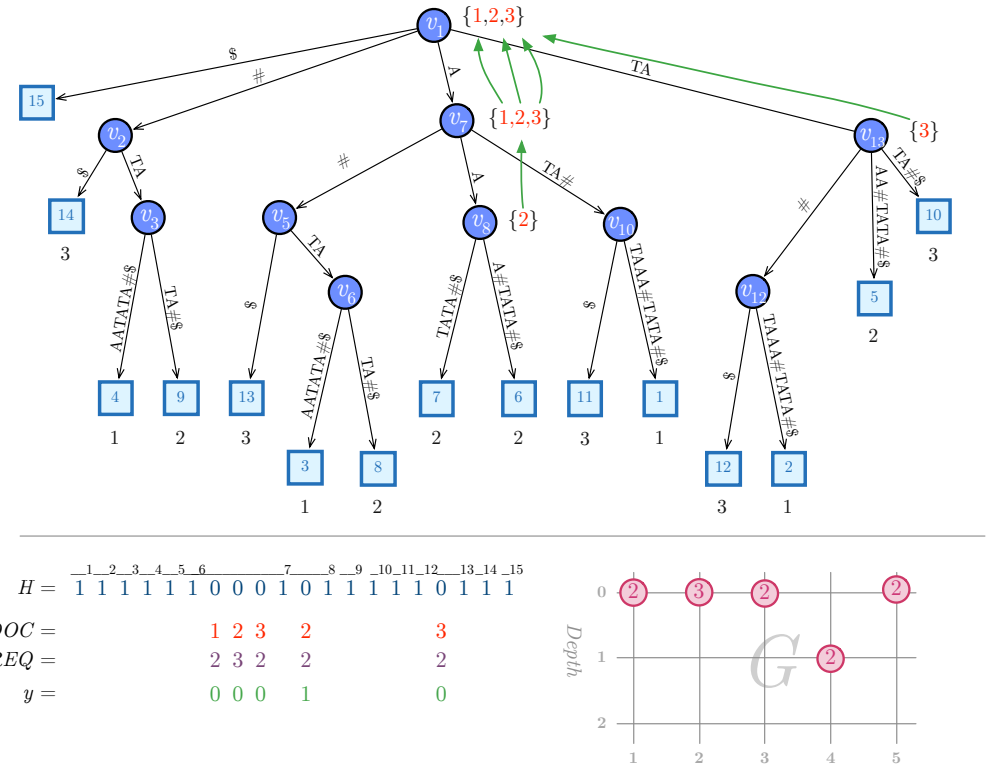


Fig. 9. Top: Suffix tree of Hon et al.'s solution with a canonical naming of the inner nodes. Each node v has attached its F-list, which excludes pointers from leaves. Bottom left: Bitvector H is built by concatenating the unary encoding of the number of pointers in the F-list of each node. Bottom right: Resulting grid.

Because we do not represent the suffix tree topology, we cannot compute the depth of the locus node v, $depth(v)$. Instead of node depths, we will store the string depths of the nodes, $|path(v)|$, as the y-coordinates. Thus the query can use simply $|P|-1$ as the y-coordinate limit of the 3-sided query. That is, after applying the mapping using H, we find the k heaviest points in the range $[sp', ep'] \times [0, |P|-1]$ of G. The top-k algorithm using this mapping is considerably simplified, as can be seen in Algorithm 3.

6.2. Smaller grid representations

In the basic implementation (WTRMQ), the grid G is represented with a combination of a wavelet tree and range maximum query (RMQ) data structures, that ensure query time $O((k + \log\log n)(\log\log n + \log k))$ if the suffix tree is of height $O(\log n)$. The price of this guaranteed worst-case time is a heavy representation of the grid, which triples the space of the basic wavelet tree of the y-coordinates vector. In addition, it has to store the vector FREQ in absolute form. We consider two ways of reducing this space.

K^2Treap^H. We can use for G a K^2-treap, which as explained is a representation that compresses the coordinates and FREQ. Even if the grid is not square as we described it, we complete it to a square grid (the added areas having no points). The price of this compression is that there are no good worst-case search time guarantees. However, the experiments will show that this grid representation uses less space and yields a query time comparable with that of the WTRMQ implementation.

ALGORITHM 3: Top-k algorithm using H-mapping index

Input: Query pattern P, amount k of documents to be retrieved
Output: top-k documents

```
00  TOPK_HMAP(P, k)
01    result ← ∅
02    [sp, ep] ←LOCATE(P)
03    [sp', ep'] ← [1, 0]
04    if sp > 1 then
05      sp' ←SELECT_1(H, sp − 1) − (sp − 1)
06    if ep > 1
05      ep' ←SELECT_1(H, ep − 1) − ep
06    result ← TOPK_GRID_QUERY(sp', ep', 0, |P| − 1, k)
07    if |result| < k then
08      result ← DOCUMENT_LISTING(sp, ep, result, k, sp)
09    return result
```

WT1RMQ^H. We can use the wavelet tree but store only one RMQ structure in it, aligned to the level of the leaves (that is, it is the RMQ of FREQ). Therefore, instead of tripling the wavelet tree space, we only add $2n$ bits. The price is that, instead of solving the top-k query on the $O(\log r)$ maximal nodes that cover $[y_1, y_2]$, we must project them to the leaves before starting the process of filling the priority queue. Therefore, the $\log r$ factor in the cost may rise to r. In exchange, since the nodes are already leaves, finding the document identifiers and weights costs $O(1)$. The overall impact is not that high if the strings $path(v)$ are of length $O(\log n)$ as before, since then $r = O(\log n)$ and the cost of this part becomes $O((k + \log n) \log k)$.

6.3. Efficient construction for large collections

A bottleneck not addressed by the WTRMQ implementation is the efficient construction of the index[1]. Both time-and space- efficiency have to be considered. We will concentrate here on the construction of H, the grid G, and its K^2-treap representation, suffix tree (CST) of $\mathcal{C}$ and a wavelet tree over the document array, WTD, of $n \log D$ bits. We perform a depth-first-search traversal on the CST and calculate, for each node v_j, the list of its document id marks, by intersecting the document array ranges of v's children. This can be done using the intersection on wavelet trees [Gagie et al. 2012]. Since we can calculate the nodes in the order of their names, we can write H and the document ids (DOC) directly to disk. In a second traversal we calculate the pointers. For each document i we use a stack to maintain the string depth of the last occurrence of i in the tree. For a node v marked with i we push the string depth of v at the first visit and pop it after all the subtree of v is traversed. Note that this time we can read H and DOC from disk (in streamed mode) and avoid the intersection. In the same traversal we can calculate the weight array FREQ by performing counting queries on WTD. Again, arrays y and FREQ can be streamed to disk.

Finally, the K^2-treap is constructed in-place by a top-down level-by-level process. Let the input be stored as a sequence of triples (x, y, w) and let 1 be the root level and b be the bottom level. First, we determine the heaviest element by a linear scan, stream its weight out to disk and mark the element as deleted. We then partition the elements of the root level into K^2 ranges, such that all elements in range r ($0 \le r \le K^2$) have

[1]For example, the liner-time LCA-based index construction method takes 1.5 hours for an 80MB Wikipedia collection, where their peak main memory usage is 12.25 GB.

the property that $x/K^{b-\ell} \bmod K = r \bmod K$ and $y/K^{b-\ell} \bmod K = r/K$. For each nonempty (resp. empty) range r we add a 1 (resp. 0) to the bitvector T, which is streamed to disk. On the next level $\ell - 1$ we can detect nodes by checking the partitioning condition of the last level, find and mark the maximum weighted entry and apply the partitioning in the node. The time complexity of the process is $O(K^2 \log_K n)$ and does not use extra space.

6.4. Summing up

We introduce new indexes, K^2Treap^{H}and WT1RMQ^H, that simplify WTRMQ by converting its MAP and TREE components into a single bitmap H. In addition, they reduce the space of the grid representation, by trading it for weaker time guarantees. The experiments will show that this does not translate into markedly reduced performance.

7. EXPERIMENTAL SETUP AND IMPLEMENTATIONS

In this section we describe the experimental setup in terms of the collections in which the indexes were evaluated, query generation and environment employed. We also explain the engineering details required to implement the indexes.

7.1. Datasets and test environment

We will split the collections into two categories depending on the alphabet type: character alphabet and word alphabet. For the character alphabet, we have four "small" collections, that have been used as baselines in previous work [Navarro et al. 2014b], and a "big" collection containing natural language text. We describe and label the collections as follows:

— KGS^c. Consists of a collection of 18,839 sgf-formmated Go game records from year 2009, containing 52,721,176 characters.
— PROTEINS^c. A collection of 143,244 sequences of Human and Mouse proteins, containing 59,103,058 symbols.
— ENWIKI-SML^c. A sample of a Wikipedia dump, consisting of 4,390 English articles, containing 68,210,334 symbols.
— DNA^c. A sequence of 10,000 highly repetitive (0.05% difference between documents) synthetic DNA sequences with 100,030,016 bases in total.
— ENWIKI-BIG^c. A bigger sample of English Wikipedia articles, consisting of 8.5GB of natural text and 3.8 million documents.

In the case of word alphabets we use three collections of documents containing natural language. We preprocessed these collections using the Indri search engine (http://www.lemurproject.org/indri/) for generating a sequence of stemmed words and excluding all html tags. The set of σ distinct stemmed words is then mapped to integers in $[1, \sigma]$.

— ENWIKI-SML^w. A word parsing of ENWIKI-SML^c, containing 281,577 distinct words.
— ENWIKI-BIG^w. A collection containing 8,289,354 words obtained from regarding the text as a sequence of words from ENWIKI-BIG^c.
— GOV2^w. Probably the most frequently used natural text collection for comparing efficiency of IR systems. The parsed collection consists of more than 72GB of data, around 40 million terms and more than 25 million documents.

Table II summarizes properties of the collections that are used. For queries, we selected 40,000 random substrings of length m obtained from the top-k documents for $m = 5$. We increase k exponentially from 1 to 256.

Table II. Collection statistics: n is the number of characters or words, D the number of documents, n/D the average document length, σ the alphabet size, $\mathcal{C}$ and total size in MB assuming that the character based collections use one byte per symbol and the word based ones use $\lceil \log \sigma \rceil$ bits per symbol.

Collection	n	D	n/D	σ	$\lvert\mathcal{C}\rvert$ in MB
character alphabet					
KGS^c	52,721,176	18,839	2798	75	51
PROTEINS^c	57,144,040	143,244	412	40	56
ENWIKI-SML^c	68,210,334	4,390	15,538	206	65
DNA^c	100,020,016	10,000	1002	4	97
ENWIKI-BIG^c	8,945,231,276	3,903,703	2,291	211	8,535
word alphabet					
ENWIKI-SML^w	12,741,343	4,390	2,902	281,577	29
ENWIKI-BIG^w	1,690,724,944	3,903,703	433	8,289,354	4,646
GOV2^w	23,468,782,575	25,205,179	931	39,177,922	72,740

For all experimental comparisons, the relevance measure used is the term frequency and the query is a single pattern string. Our implementation and benchmarks are publicly available at https://github.com/rkonow/surf/tree/wt_topk.
All experiments were run on a server equipped with an Intel(R) Xeon(R) E5-4640 CPU clocking at 2.40GHz. All experiments use a single core and at most 150GB of the installed 512GB of RAM. All programs were compiled with optimizations using g++ version 4.9.0. The test collections are all available at http://algo2.iti.kit.edu/gog/projects/ALENEX15/collections. The only exception is GOV2, which is not free and can be obtained from http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm.

7.2. Baselines

As baselines we use the GREEDY solution [Culpepper et al. 2010] and the SORT approach [Gog et al. 2014], which is included in the Succinct Data Structure Library (SDSL, https://github.com/simongog/sdsl-lite). Baseline SORT uses the CSA and document array DA. It first gets the range $[sp, ep]$ of all occurrences of P from the CSA. Then it copies all documents from $\mathrm{DA}[sp, ep]$ and extracts the top-k most relevant documents by sorting and accumulating the occurrences. Baseline GREEDY also gets the range $[sp, ep]$ from the CSA, but then does a greedy traversal of the wavelet tree over DA to get the top-k documents. As an additional baseline we also ran the experiments using the original source code of Navarro et al. [2014b] (NPV). This implementation consists of more than 24 alternative configurations, including grammar compressed wavelet trees using different bitmap representations and improvements to the GREEDY algorithm. For the experimental time-related results, we will always show the points that correspond to the pareto-optimal border in terms of space and time considering all possible configurations. We refer to this index as as NPV^{opt}. We will not compare to the work of Patil et al. [2011] since the space requirements of their solution is considerably bigger (see Table I) than the ones that are being evaluated in this work.

7.3. Implementation of WTRMQ

The implementation of WTRMQ requires the assembly of a complex set of compact data structures: CSA, wavelet tree (WT), RMQ, rank/select-capable bitmaps, direct access codes, compact tree topologies and an efficient integer array representation. In practice, for the CSA, we use an off-the-shelf SSA from *PizzaChili* site, (`http://pizzachili.dcc.uchile.cl`), and add a rank/select capable bitmap, `BitSequenceRG`, implemented in the LIBCDS library (https://github.com/fclaude/

libcds), that requires 5% of extra space. Bitmap DOCBITMAP is used to mark where each distinct document starts in $\mathcal{C}$. This way, the document corresponding to $\mathrm{SA}[i]$ is obtained by performing $\mathrm{RANK}_1(\mathrm{DOCBITMAP}, \mathrm{SA}[i])$. We also use the 5% space overhead bitmaps for the MAP bimap $B[1, 2n]$. We use the fully-functional compact tree representation [Arroyuelo et al. 2010] requiring 2.3 bits per element to perform LCA and PREORDER_SELECT over the suffix tree topology. For LEAF_RANK and LEAF_SELECT we use another bitmap $L[1, 2n]$ where the leaves are marked, and we use 5% space overhead to implement RANK_1 and SELECT_1 (this could be slightly improved but makes little difference). We employ the original optimal implementation of direct access codes for representing the weights. For the wavelet tree, we use the no-pointers version from LIBCDS (`WaveletTreeNoPtrs`) using `BitSequenceRG` to represent the bitmaps. Recall that the wavelet tree has to be enhanced to support range maximum queries (RMQ) at each of the weight sequences $W(v_i)$. We implemented the RMQ structure that requires 2.3 bits per element [Fischer and Heun 2011]. For the document identifiers we use an integer array that requires $\lceil \log(D + 1) \rceil$ bits per element and access any position directly. All of these implementations and data structures were limited to handle $2^{32} - 1$ memory addresses, thus this index is not capable of handling big datasets such as ENWIKI-BIG^c.

7.4. Implementation of K^2Treap^H

The set of data structures required for implementing K^2Treap^H is smaller: We need a CSA, RMQ, rank/select capable bitmaps, direct access codes, a K^2-treap, and an efficient integer array representation. We based our implementation on structures from SDSL and will use the SDSL class names in the following. For character based indexes, we use a CSA based on a wavelet tree (`csa_wt`) which is parametrized by a Huffman-shaped wavelet tree (`wt_huff`) which uses a compressed bitmap (`rrr_vector<63>`). For word based indexes, we opted for a CSA based on the Ψ function (`csa_sada`) which is compressed with Elias-δ codes (`coder::elias_delta`). The latter provided a better time-space trade-off for large alphabet pattern matching. For the mapping we store the bitmap H using a compressed bitmap `rrr_vector<63>`. The SELECT_1 performance of `rrr_vector<63>` is slow compared to a plain bitmap representation but negligible in our case, where only two SELECT_1 queries are done per top-k query (recall Algorithm 3). The RMQC structure is realized by a MinMax-tree implementation `rmq_succinct_sct` which uses about 2.3 bits per element. The document ids, DOC, are stored in an integer vector (`int_vector`) using $\lceil \log(D + 1) \rceil$ bits per element. For this work we have implemented the K^2-treap structure and added it to SDSL (available now as `k2_treap`). The K^2-treap implementation is generic (the value K, the bitmap representation of the K^2-ary tree, and the representation of the vector of relative weights, can be parametrized) and complements the description of Brisaboa et al. [2016] with an efficient in-place construction described in Section 6.3. For the relative weights we use direct access codes (`dac_vector<4>`) with a fixed width of 4.

7.5. Implementation of WT1RMQ^H

The only difference between K^2Treap^H and WT1RMQ^H is the grid implementation. The WT1RMQ^H solution uses a wavelet tree and an RMQ data structure to represent the grid. We opted for a `wt_int` parametrized with `rrr_vector<63>`. The compressed bitvector decreased the size considerably since the grid contains many small y-values. The absolute weights are stored in a `dac_vector<4>` of fixed width 4, which gave the most practical result when performing parameter tuning experiments. The RMQ structure was again realized by `rmq_succinct_sct` using 2.3 bits per element.

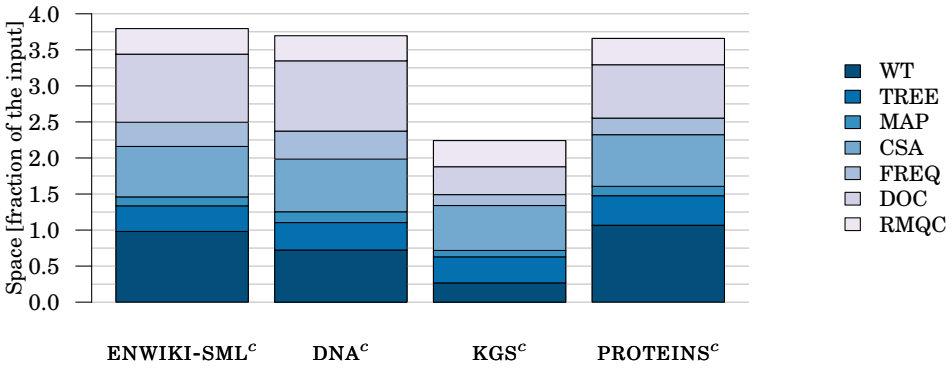


Fig. 10. Space usage decomposition as fraction of the input, for each structure employed in WTRMQ. DOCBITMAP is accounted for inside RMQC.

8. RESULTS

In this section we study the practical properties of the different index implementations. We first analyze the space consumption of the structures; both total space and the space of substructures – like the grid – are considered. Then the query time is analyzed. Again, we consider total query time and also study the time for the different phases of the query process. To get a precise picture about the performance we vary various parameters: the number of results k, the pattern length m, and collections. The latter are from different application domains, cover different scales of magnitude – from 50 MB to 72 GB –, and vary in alphabets size from 4 to almost 40 million. Note that some non-SDSL based baselines did not support alphabets larger than 256 and collection sizes above 2 GB and were therefore only evaluated in some setups.

8.1. Space

We start by decomposing the space required of our first index, WTRMQ. Fig. 10 shows the space required for each of its components in terms of fraction of the input. For most collections the space requirements are quite similar, except for the KGS^c collection. In general, WTRMQ requires 2–4 times the size of the input. If we analyze the space of the components, the most resource-consuming piece is the wavelet tree with multiple RMQ structures, which requires space up to the size of the collection. The CSA requires 0.6–0.7 times the collection size. Recall that for this implementation we need to store the topology of the suffix tree (TREE) plus two bitmaps to perform the mapping of the suffix tree nodes to the grid (MAP). These data structures add up to 0.6 times the size of the input. Storing the frequencies (FREQ) using direct access codes uses 0.2–0.3 times the size of the input, which is about the same size of the tree topology. The space requirements to represent WTRMQ is probably impractical for real scenarios. Furthermore, the implementation of this index is not able to handle collections that are bigger than 200MB nor collections that are parsed as words.

Fig. 11 shows the decomposition of the index $K^2\text{Treap}^H$, which employs the H bitmap to map the suffix tree nodes to the grid. In addition, this index uses the K^2-treap to represent the grid G. The efficient representation of G is about 30% smaller than WTRMQ. We exemplify the space reduction of $K^2\text{Treap}^H$ compared to WTRMQ in the case of ENWIKI-SML^c. While the grid mapping takes 46.5 MB ($8.6 + 12.8 + 25.1$ for bitvector B, L, and the tree topology) in WTRMQ, $K^2\text{Treap}^H$ just takes 6.8 MB for bitvector H (13.0 MB in uncompressed form). The grid representation as a K^2-treap takes 57.7 MB ($21.7 + 24.5 + 11.5$ for weights, y-values, and topology) compared

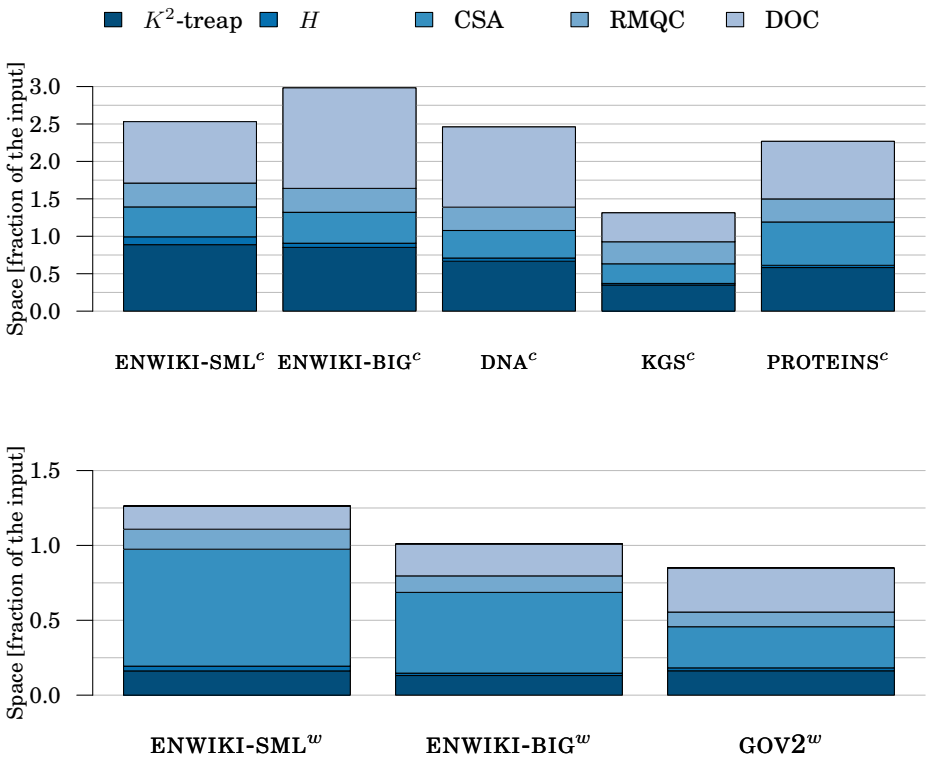


Fig. 11. Space usage decomposition as fraction of the input, for each structure employed in K^2Treap^H. Top: results for character alphabet collections. Bottom: results for word alphabet collections.

to 77.7 MB (23.3 MB for weights and 55.4 MB for the RMQ-enhanced wavelet tree) in WTRMQ. These space savings result in index sizes between 1.5–3.0 times the original collection size for character alphabet collections, see Fig. 11 top. For word-parsed collections, IDX_GN takes space close to the original input, for example, for the 71.0 GB word parsing of GOV2^w, the index size is 64.6 GB. Recall that this space includes the CSA component, which can recover any portion of the text collection, and thus the collection does not need to be separately stored.

The difference between K^2Treap^H and WT1RMQ^H is the grid representation, so we compare the space required by the bit-compressed wavelet tree with a single RMQ structure to the space required of the K^2-treap (see Fig. 12). In this case, the y-axis represents the fraction of the space required for the wavelet tree-based grid to the space required to represent the grid using the K^2-treap. We observe that, in most of the character based collections, except for the case of DNA^c, the space savings of the wavelet tree are 30% to 40%. A similar result can be seen in the case of word collections.

Fig. 13 shows the space comparison in terms of fraction of the input of all of the introduced indexes and the baselines for the character alphabet collections. ENWIKI-BIG^c is not shown due to the implementation constraints of some of the indexes. As previously mentioned, NPV^{opt} chooses the best possible result obtained from all the alternative configurations of the implementations presented by Navarro et al. [2014b]. In this case, we show the variants that yield the best compression for each collection (NPV^{min}) and the ones yielding the highest space usage (NPV^{max}). We also show the space requirements of the other baselines, GREEDY and SORT. For almost all cases WTRMQ is

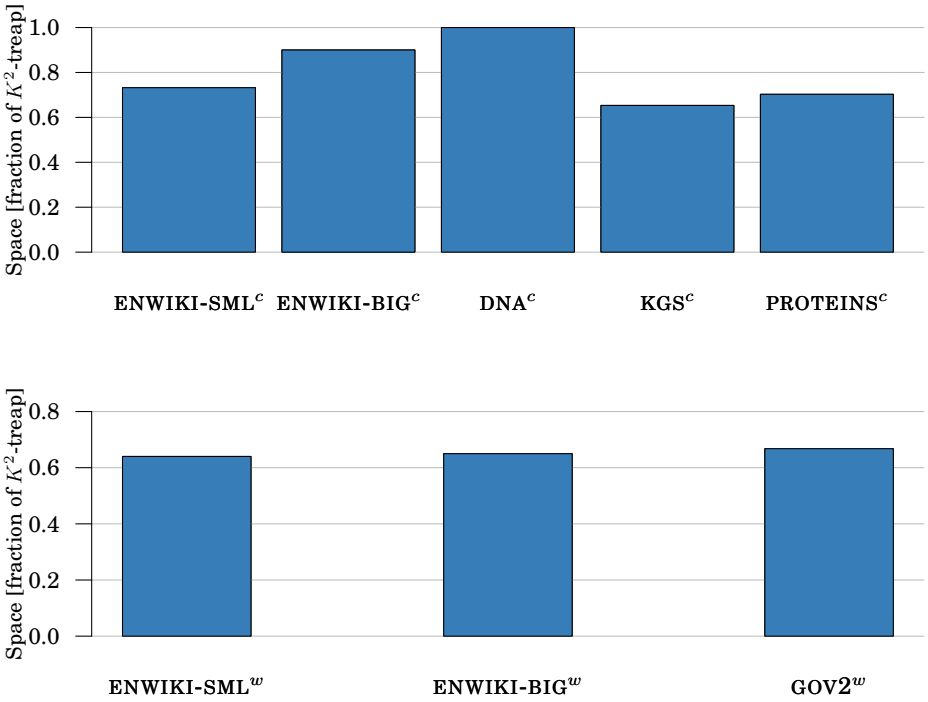


Fig. 12. Comparison of the space required to represent the grid G using a wavelet tree plus a single level of RMQ (WT1RMQ^H) in terms of the space required if the K^2-treap is used.

the most space-demanding index, even when compared to the uncompressed version NPV^{max}. Only for the KGS^c collection is WTRMQ smaller than NPV^{max}. WT1RMQ^H and K^2Treap^H require about 20% more space than other variants for ENWIKI-SML^c and DNA^c. However, the space of these indexes for KGS^c is about half of GREEDY and SORT, and they are smaller than the most compressed solution, NPV^{opt}.

8.2. Retrieval speed when varying k

The time of a top-k query consists of the time to match the pattern in the CSA, the time to map $[sp, ep]$ to the grid, the time to report the top-k documents using the K^2-treap or wavelet tree and, if less than k documents are found, the time to report frequency-one documents (which are not stored in the grid) using RMQC and, again, the CSA. The top-k retrieval time for a pattern P depends on multiple factors: First, the length m of P. For an FM-index based CSA (resp. Ψ-function based) it takes $O(m \log \sigma)$ (resp. $O(n \log n)$) steps. Second, the time for the mapping from the lexicographic range into the grid's x-range. This method, described in Algorithm 3, requires two SELECT$_1$ operations, whose time is negligible compared to the first step. The last steps depend on the output size of the query, k, compared to the number of occurrences of the pattern. Documents in which the pattern occurs more than once are calculated via the corresponding top-k grid query (either K^2-treap for K^2Treap^H or wavelet tree for WTRMQ and WT1RMQ^H), while documents in which the pattern just occurs once are retrieved via the document listing algorithm that uses the RMQC structure on CA, a locate oper-

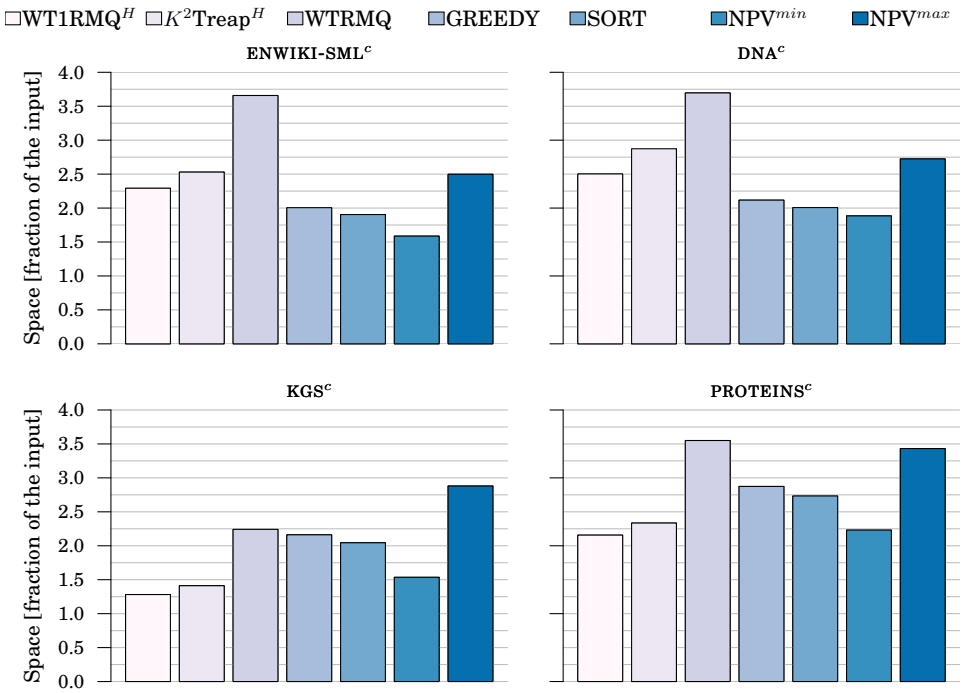


Fig. 13. Space as fraction of the input for small character alphabet collections. NPV^{min} is the smallest possible result obtained from Navarro et al. [2014b], while NPV^{max} corresponds to the uncompressed variant.

ation on the CSA, and a RANK$_1$ on DOCBITMAP. We examine the cost of the different phases in Fig.14 for K^2Treap^H.

As expected, the pattern matching with the CSA is independent of k and takes about 5–10μs. The time to retrieve the first document out of the K^2-treap is relatively expensive. For most of the collections it is about 40–70μs and is dominated by the cost of the priority-queue based search down the K^2-treap until a first (heaviest) element within the query range is found.

The subsequent documents are cheaper to report. The time spent in the K^2-treap to report 16 documents is about twice the time to report a single document except for the case of PROTEINS^c. The average time per document retrieved via the K^2-treap is typically about 3–5μs for $k \geq 64$. Essentially, for each such document, one must perform a constant number of RMQ operations and extract a suffix array cell from the CSA. The cost of an RMQ is typically below 2μs [Gog 2011, Sec. 6.2], while the CSA access accounts for the remaining 100–300μs. The CSA access time is linearly dependent on a space/time tradeoff parameter s, which is set to $s = 32$. Note that top-k queries are meaningful when documents have different weights, and thus large k values that retrieve many documents with frequency 1 are not really interesting. An interesting case arises for the DNA^c collection, where the amount of time spent to retrieve documents with frequency 1 is negligible. Recall that DNA^c contains synthetic highly repetitive sequences, and a limited alphabet size $\sigma = 4$, therefore it is quite improbable to generate a query-string of length $m = 5$ that has a single occurrence.

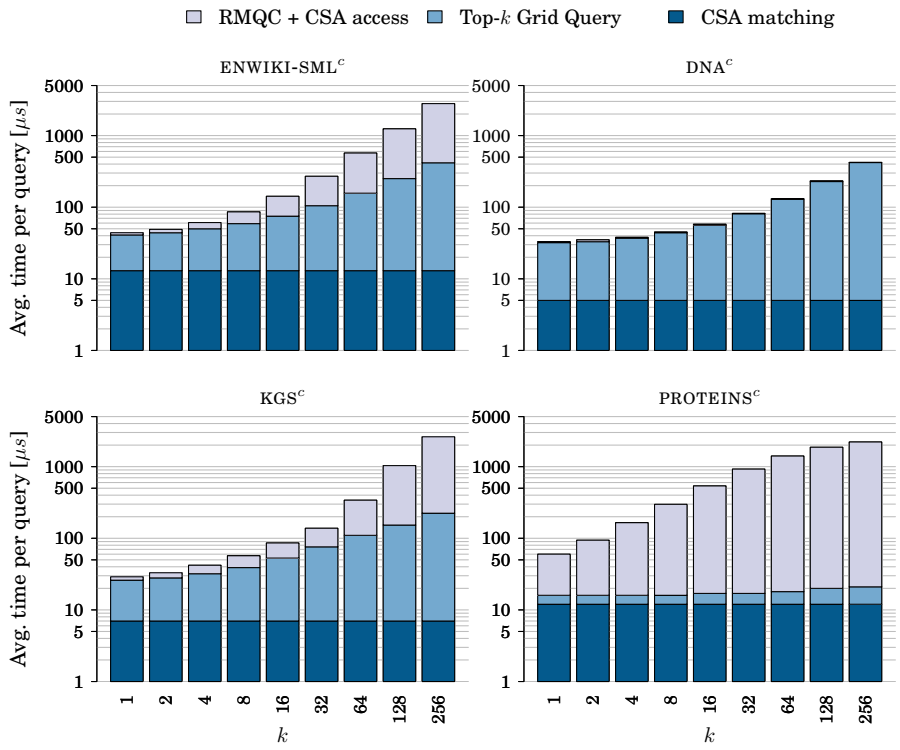


Fig. 14. Detailed breakdown of average query time for index K^2Treap^H which uses a CSA pattern search, a K^2-treap for the top-k grid query, and the document listing algorithm that requires RMQ queries plus CSA accesses (time fractions plotted from bottom to top in log-scale). We use $m = 5$.

Fig. 15 shows the results for our biggest character alphabet dataset, ENWIKI-BIG^c. On the left part of the figure we show the time required to perform each part of the query process as in Fig. 14. On the right, we show average time per document retrieved and perform a breakdown depending on the mechanism that was employed. For values of $k \leq 8$, the weighted average amount of time spent to retrieve each document is always greater than $10\mu s$. As the value of k increases, the average time to report a document decreases. For $k = 256$ the average time required is less than $5\mu s$. Note that for all k values (for $k \geq 4$, since for smaller k values, it was not necessary to execute the single-occurrence procedure) the time for the RMQC+CSA accesses to report a document is about 60–90μs.

We now examine the performance of the K^2-treap grid representation compared to the use of the wavelet tree alternative. We use the implementation of WT1RMQ^H using compressed bitmap representations and only one RMQ level, since the results obtained with WTRMQ were almost identical, except for PROTEINS^c where WTRMQ is 3 times *slower* due to the many RMQs performed. Fig. 16 shows the comparison of the average time required to complete a top-k query, for varying k, using the K^2-treap or the wavelet tree. For most cases, the wavelet tree-based approach is faster than the K^2-treap, except in the DNA^c collection, where for $k \geq 8$ the K^2-treap is faster. The most significant difference can be seen on ENWIKI-SML^c, where on average the wavelet tree is two times faster. Fig. 17 shows the times on our biggest character alphabet collection (ENWIKI-BIG^c): the wavelet tree is about 50% slower for $k \leq 64$. This difference gets

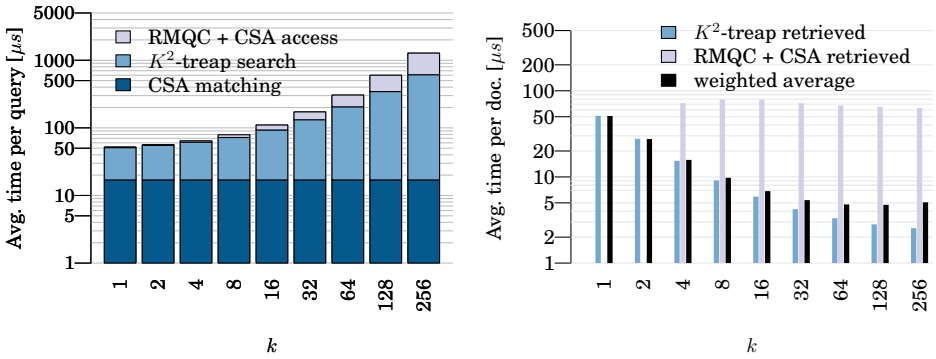


Fig. 15. Query times for IDX_GN on ENWIKI-BIG^c, with $m = 5$. Left: Query time depending on k with a detailed breakdown of the three query phases. Right: Average time per document, considering those retrieved from the K^2-treap, with RMQC+CSA, and their (weighted) average. The CSA matching time is included in all cases.

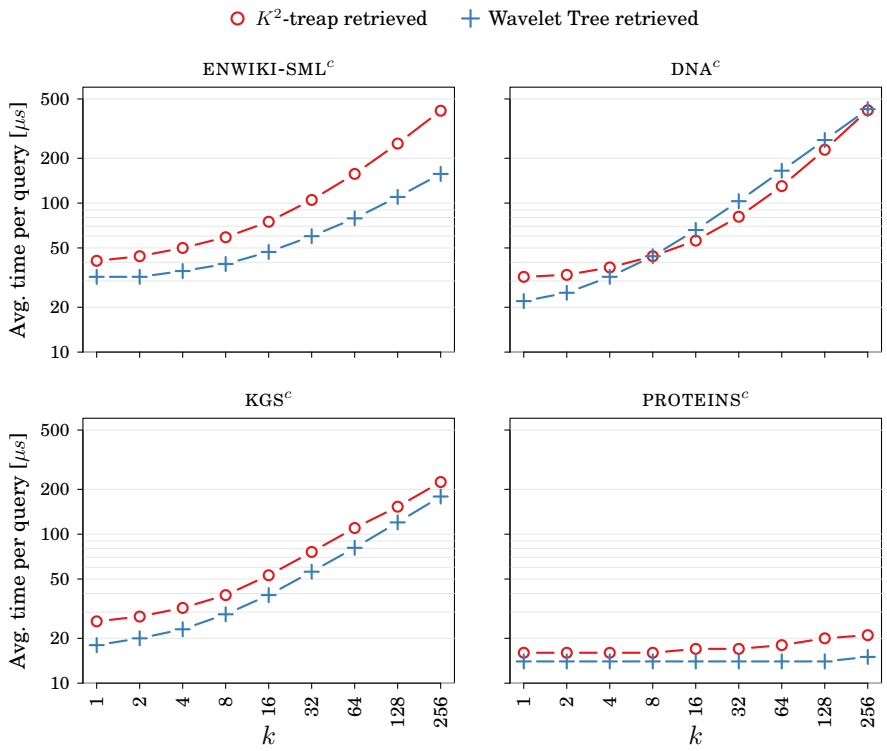


Fig. 16. Comparison of the average time per query required to retrieve the top-k results using a K^2-treap or the wavelet tree for representing G.

smaller for larger k values, and for $k = 256$ the wavelet tree is already 10% faster than the K^2-treap. This is quite different from the results obtained on the small collection of the same type (ENWIKI-SML^c), where the wavelet tree was up to 5 times faster for $k = 256$ and dominated the average time per query for all the k values.

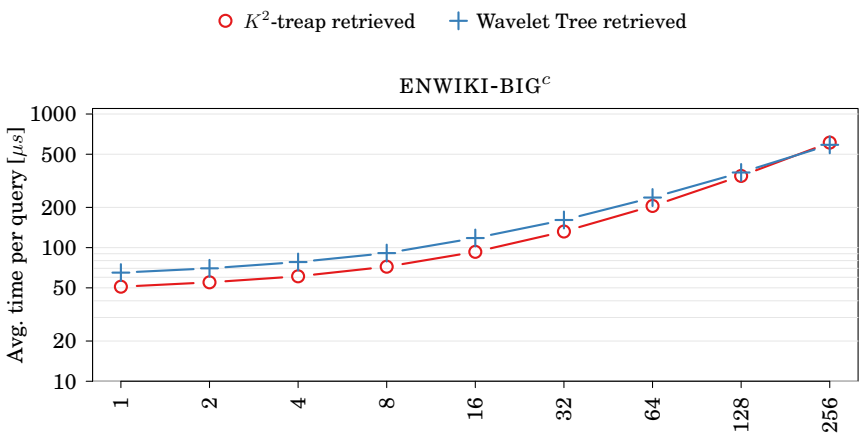


Fig. 17. Comparison of the average time per query required to retrieve the top-k results using a K^2-treap or the wavelet tree for representing G for ENWIKI-BIG^c.

We compare the overall performance of our indexes against the baselines in Fig. 18. Recall that index NPV^{opt} represents the best achieved result from all alternatives in terms of space/time. We show the results using the smaller character alphabet collection since there are implementation constraints for NPV^{opt} and WTRMQ. Except for the PROTEINS^c collection, all of our indexes require less than $200\mu s$ for $k \leq 16$. We start by analyzing the performance for the English Wikipedia collection (ENWIKI-SML^c). Our indexes are faster than all other approaches for $k \leq 32$. For larger k values, the best combination of NPV^{opt} is faster than our approaches by up to a factor of 3 ($k = 256$). In the case of the DNA^c collection, all of our indexes are up to 12 times faster than the fastest alternative (NPV^{opt}). As mentioned before, the performance of WTRMQ is similar to that of WT1RMQ^H and K^2Treap^H. The extra RMQ operations performed at each wavelet tree level add a constant time factor to the query time, as can be clearly observed from the results on DNA^c and PROTEINS^c. In the case of KGS^c, the difference between our indexes and the other approaches is considerably bigger: our approaches are 21 times faster than the closest alternative (NPV^{opt}) for $k = 2$, and even for larger k values all of our approaches are up to 3 times faster than the fastest baseline. Finally, in the case of PROTEINS^c, our results show that there is no alternative that is faster than the most basic approach, which is sorting (SORT). In this case, our indexes are slower than SORT for all k values, and also than NPV^{opt} for $k \geq 4$. The constant factor added to the query times for the extra RMQ operations of WTRMQ is more evident for this collection, making this alternative up to 10 times slower than WT1RMQ^H and K^2Treap^H. Note that in this collection most patterns occur once in each document, thus top-k queries are equivalent to plain document listing queries.

Fig. 19 shows the results for the bigger character based collection (ENWIKI-BIG^c). Note that for this experiment we are not able to construct the index using NPV^{opt} and WTRMQ since their implementations are not able to handle more than $2^{32} - 1$ memory addresses. For all distinct k values, we are up to 100 times faster than the fastest baseline (GREEDY) and even 1,000 times faster than the simple SORT method. The reason for this is that GREEDY, as well as SORT, have to handle larger $[sp, ep]$-intervals for this bigger collection, while our index is less dependent on this factor. As mentioned before, it turns out that for the case of the collection ENWIKI-BIG^c WT1RMQ^H is slower

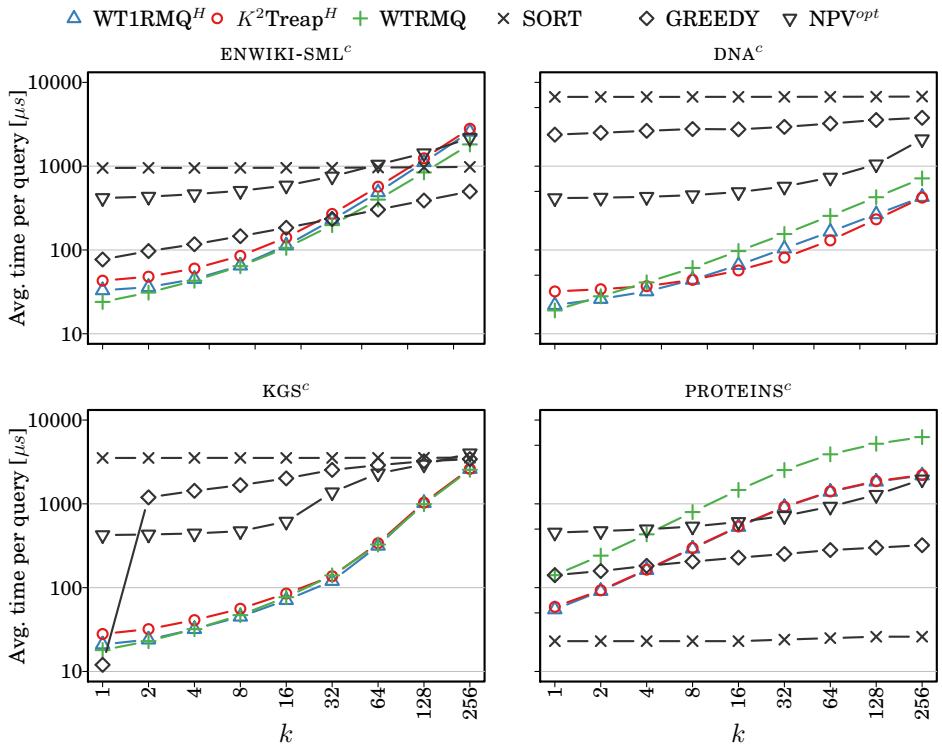


Fig. 18. Average time per query, in microseconds, for different k values and fixed pattern length $m = 5$, evaluated on small character alphabet collections.

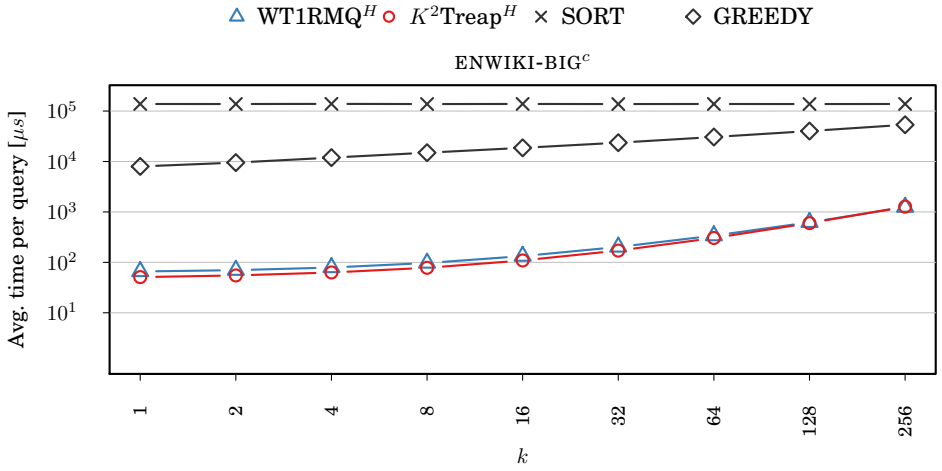


Fig. 19. Average time per query in microseconds for varying k values and fixed pattern length $m = 5$ using big character alphabet collections.

than K^2Treap^H for values of $k \leq 128$. Therefore, the difference between heuristics and indexes with stronger guarantees shows up as the data sizes increase.

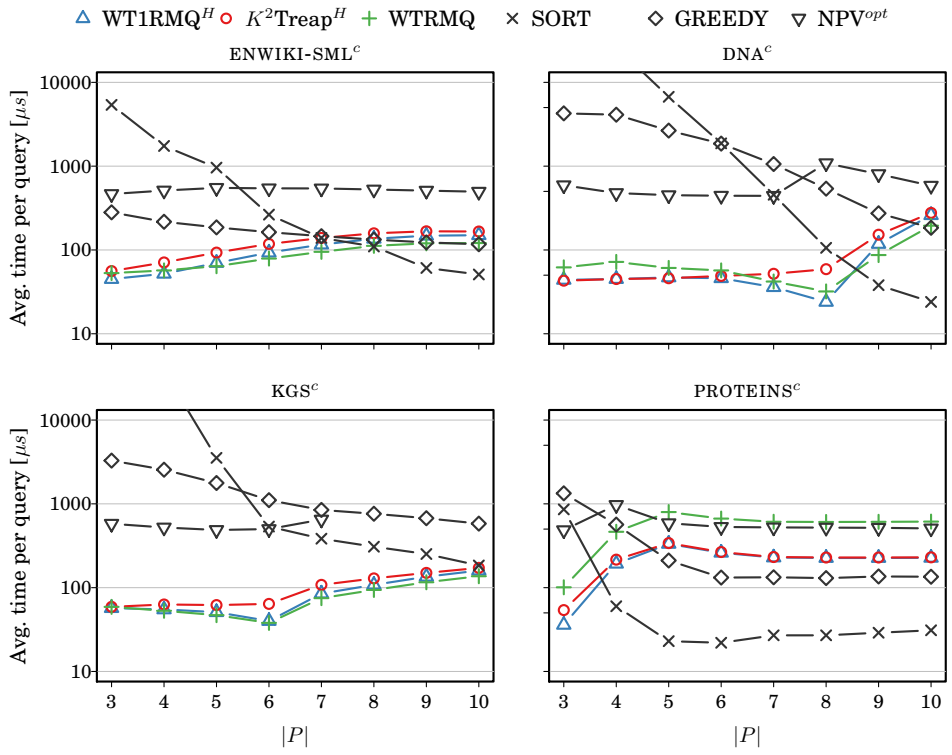


Fig. 20. Average time per query in microseconds for different pattern lengths and fixed $k = 10$ value.

8.3. Varying pattern length

We proceed to analyze the effect of changing the query pattern length m. As before, we selected 40,000 random substrings of lengths m ranging from 3 to 10 and obtained the top-k documents for fixed $k = 10$. Fig. 20 shows the results, in terms of average time per query for different query pattern lengths. For the collection ENWIKI-SML^c, our index query time ranges in 80–110μs. Note that for query patterns longer than 8 symbols, SORT is the fastest index. Since the ranges in the wavelet tree for longer patterns are smaller, GREEDY starts to be competitive for $m \geq 7$. In general, the query times are quite similar for all of our indexes on this collection. A similar scenario arises on the DNA^c collection: our indexes range in 80–180μs, and for longer patterns SORT is the best alternative. Note that NPV^{opt} is 10 times slower for most m values. In the case of KGS^c collection, our indexes are the best for the whole range of pattern lengths and we still are one order of magnitude faster than GREEDY and NPV^{opt}. A special case arises for the PROTEINS^c collection. Our indexes are generally slower than the baselines, especially WTRMQ, which is up to 5 times slower than WT1RMQ^H and K^2Treap^H. Still, they are faster than NPV^{opt}. In this case, the best alternative is the simplest solution: the index SORT, for patterns having more than 4 symbols.

We show the results of our biggest character alphabet dataset, ENWIKI-BIG^c in Fig. 21. Recall that for this case, we are not able to compare with WTRMQ and NPV^{opt} due to the implementation constraints already described before. Compared to SORT and GREEDY our indexes are up to 100 times faster for small pattern lengths and up to 8 times faster for the longest case ($m = 10$).

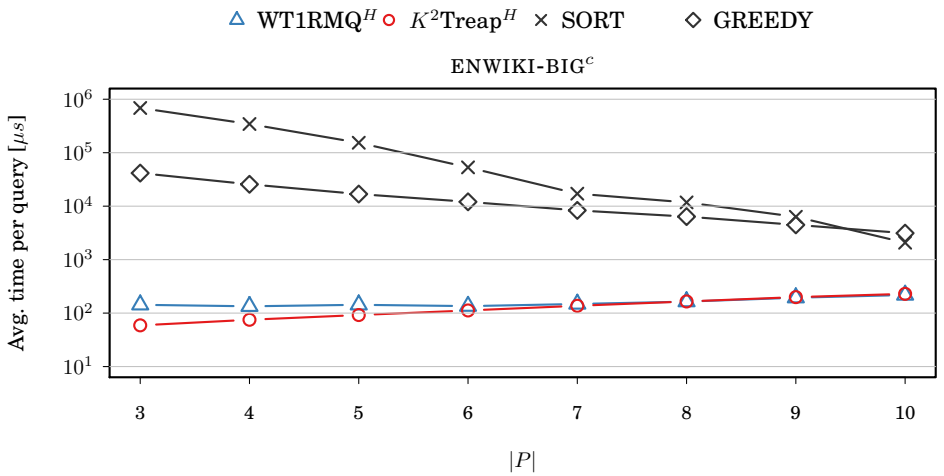


Fig. 21. Average time per query in microseconds for different pattern lengths and fixed $k = 10$ value for ENWIKI-BIG^c.

8.4. Word alphabet collections

One of the most important feature of our best implementations (K^2Treap^H and WT1RMQ^H) is that they are able to index collections with large alphabets, thus allowing the mapping of words to integer symbols. We measured the performance of our indexes and other baseline implementations on the word alphabet collections (ENWIKI-SML^w, ENWIKI-BIG^w, and GOV2^w). We used the same experimental setup, generating 40,000 single-word ($m = 1$) queries chosen at random from each collection $\mathcal{C}$ and increasing k exponentially from 1 to 256.

We start by analyzing the results with the same breakdown of the top-k procedure as done before for the character alphabet case. We show the details of the average query time required for the index K^2Treap^H on word alphabet collections in Fig.22. The pattern matching using the CSA takes less than $5\mu s$ for all cases. These results are considerably faster than those obtained on character alphabet collections. The main reason for this difference is that the CSA searches for a shorter pattern ($m = 1$ words), even if on a larger alphabet. In general the top-k grid query takes a great portion of the total time, of about 20–30μs for retrieving a single document. Interestingly, for ENWIKI-SML^w, the portion of total time spent performing the single-occurrence procedure (RMQC + CSA accesses) is much bigger than for the larger collections. This is expected, since ENWIKI-SML^w contains a small amount of document (4,390) when compared to ENWIKI-BIG^w (3,903,703) and GOV2^w (25,205,179), and thus in the latter it is more likely to find k documents where P appears more than once. In general, K^2Treap^H is up to three times faster in these types of collections than in the smaller, character alphabet ones.

We show the average time required to retrieve a document, depending on whether it was retrieved using the K^2-treap or the single-occurrence procedure. We show the results obtained for the GOV2^w collection in Fig. 23. As before, retrieving a single document using either approach is costly: about 20μs with the K^2-treap and 100μs with RMQC + CSA accesses. The time per document decreases significantly as k increases. For $k = 256$, the average time to retrieve a single document is below 2μs for the K^2-treap and 31μs for the single-occurrence procedure. As before, this is because the K^2-treap spends some time until extracting the first result, and the next ones come faster. Instead, the document listing method has a more constant-time behaviour: Af-

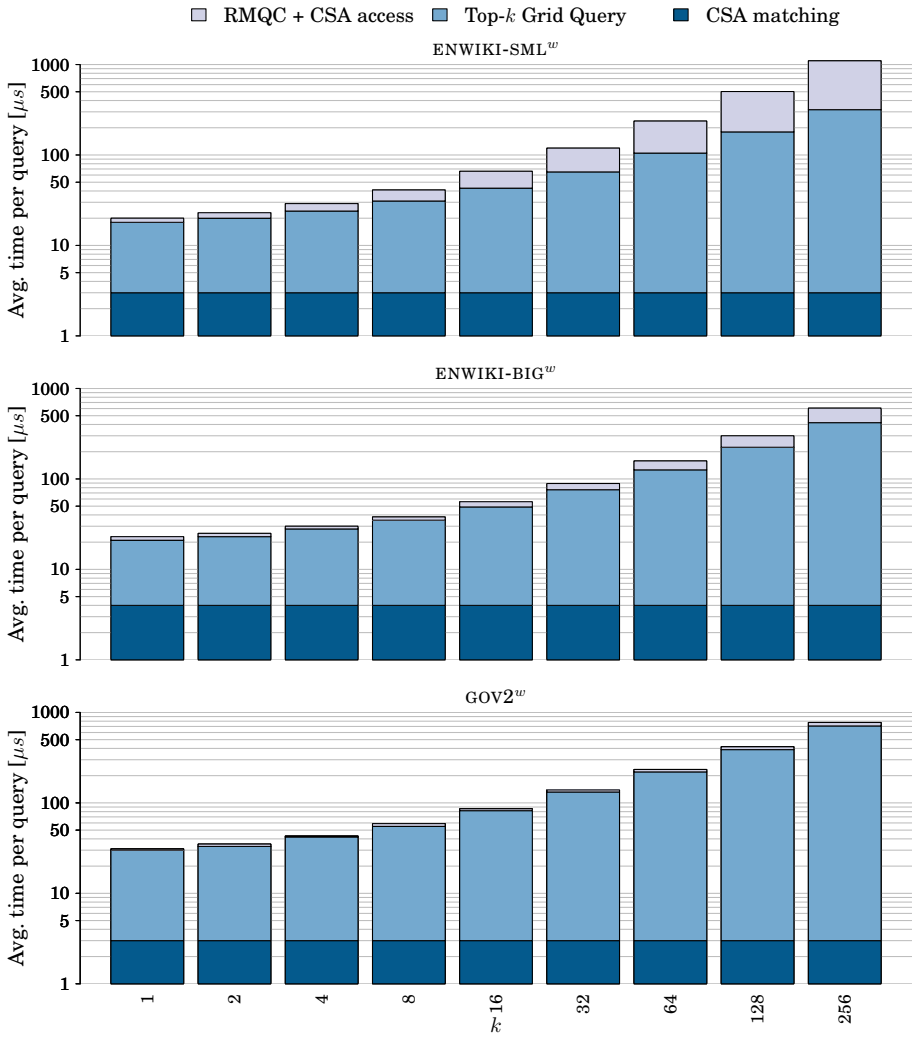


Fig. 22. Detailed breakdown of average query time for index K^2Treap^H on word alphabet collections.

ter $k \geq 4$, the RMQC + CSA retrieval time does not decrease when more documents are requested.

Fig. 24 compares the two grid representations: the K^2-treap and the wavelet tree. Interestingly, for the small collection (ENWIKI-SML^w) the K^2-treap is slower than the wavelet tree, but it is considerably faster for larger collections (ENWIKI-BIG^w and GOV2^w). In detail, for the small collection, the wavelet tree is up to twice as fast as the K^2-treap, and for the larger ones, the wavelet tree is up to twice as slow. This result is also different when compared to the character alphabet collections, where in most cases the wavelet tree is faster than the K^2-treap. This is due to the x-range of the query: since the pattern length is $m = 1$, the x-ranges in the queries bigger for the large collections. This affects negatively the wavelet tree, because the two RANK$_b$ operations used to map the interval $[x_1, x_2]$ are far apart and require separate cache

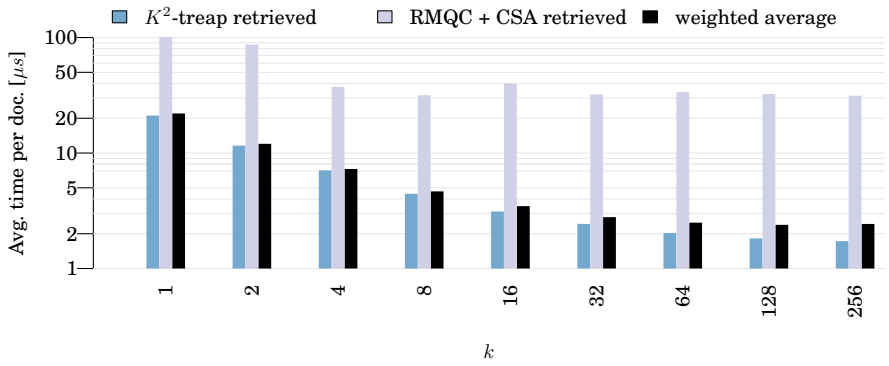


Fig. 23. Query times for IDX_GN on GOV2^w, with $m = 1$. Average time per document, considering those retrieved from the K^2-treap, with RMQC+CSA, and their (weighted) average. The CSA matching time is included in all cases.

misses. Instead, the K^2-treap has a higher chance of having the heaviest point of each subgrid inside the bigger query area, and thus it might find results sooner.

Fig. 25 shows the comparison between our indexes and the baselines (GREEDY and SORT). For the small Wikipedia collection our indexes are one order of magnitude faster than GREEDY for all values up to $k \leq 64$, taking less than $100\mu s$ on average. On the other hand, the naive SORT is up to 1,000 times slower than our indexes for small $k \leq 8$ values, and 10 times slower for $k = 256$. For the bigger texts, SORT is not considered since it required more than 5 seconds to execute. In these cases, our indexes are undisputedly the fastest alternative, being about 1,000 times faster for $k = 10$ and almost 100 times faster for the largest k value. Note that from the two alternatives, K^2Treap^H is faster than WT1RMQ^H as the grid search is considerably faster when performed on the K^2-treap than on the wavelet tree.

9. CONCLUSIONS AND FUTURE WORK

Top-k document retrieval on general string collections is a challenging problem that is not well solved with classical pattern matching indices. Since the theoretical time-optimal solution [Navarro and Nekrich 2012] uses impractical amounts of space, there has been a continued line of research aiming at engineering slower alternatives, which use 2–5 times the text size [Culpepper et al. 2010; Navarro et al. 2014b; Gog et al. 2014]. In this article we have shown that, instead, the time-optimal solution can be engineered so that it uses only 2.5–3.0 times the text size and answers queries within microseconds. This is remarkable if we consider that a naive implementation of the theoretical solution would use about 80 times the text size. Our index is typically 10 times faster than previous solutions, even than those using more space. The only indices using less space than our implementation [Navarro et al. 2014b] may be about 30% smaller, but hundreds of times slower.

We have also developed efficient construction algorithms for our index, which allow us to index hundred-gigabyte natural-language collections of word sequences, over vocabularies of million symbols. In this case, our top-k indices offer a top-k functionality analogous to that of inverted indices. In particular, they can easily answer top-k phrase queries, which are hard to handle with inverted indices. Instead, extending our index to handle weighted Boolean queries, or more complex relevance measures like BM25, is an interesting future challenge.

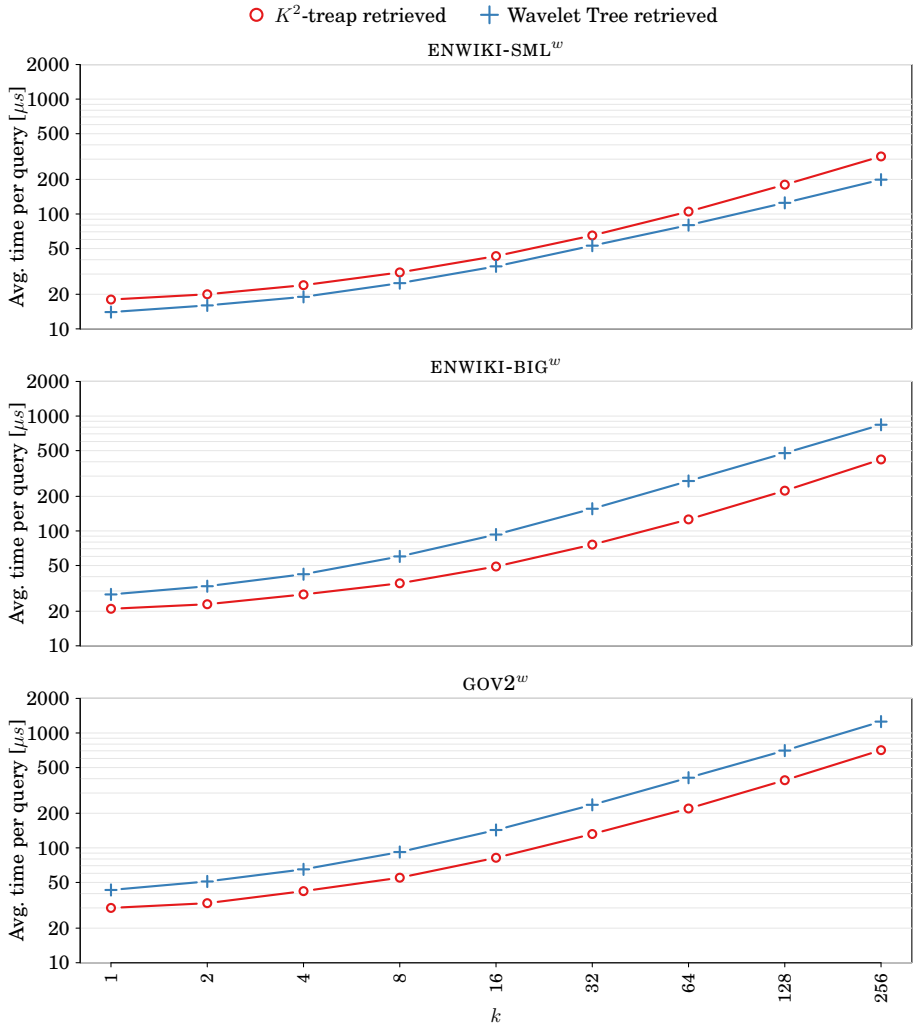


Fig. 24. Comparison of the average time per query to retrieve the top-k results using a K^2-treap or the wavelet tree for representing G in word alphabet collections.

REFERENCES

D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. 2010. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 84–97.

R. Baeza-Yates and B. Ribeiro-Neto. 2011. *Modern Information Retrieval* (2nd ed.). Addison-Wesley.

D. Belazzougui, G. Navarro, and D. Valenzuela. 2013. Improved compressed indexes for full-text document retrieval. *Journal of Discrete Algorithms* 18 (2013), 3–13.

D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. 2005. Representing trees of higher degree. *Algorithmica* 43, 4 (2005), 275–292.

N. Brisaboa, G. de Bernardo, R. Konow, G. Navarro, and D. Seco. 2016. Aggregated 2D Range Queries on Clustered Points. *Information Systems* 60 (2016), 34–49.

N. Brisaboa, S. Ladra, and G. Navarro. 2013. DACs: Bringing direct access to variable-length codes. *Information Processing and Management* 49, 1 (2013), 392–404.

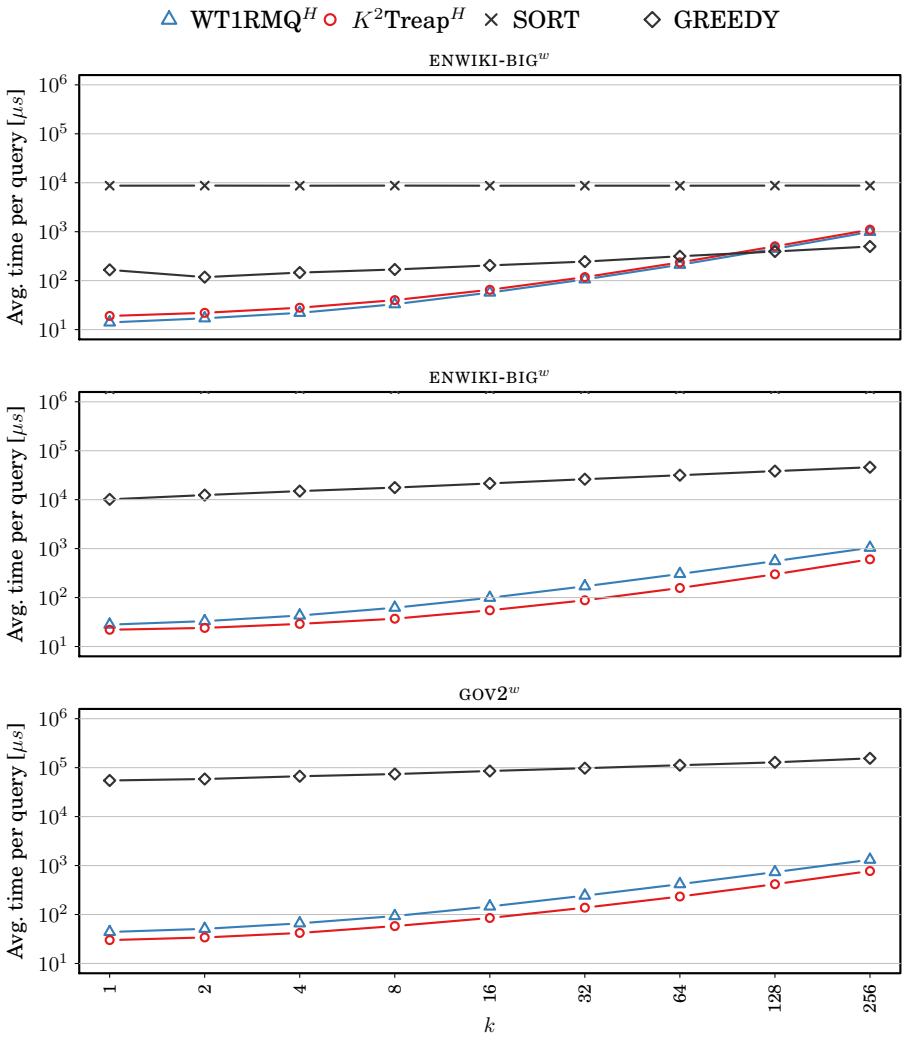


Fig. 25. Average time per query in microseconds for different k values and fixed pattern length $m = 1$ on word alphabet collections. SORT is not included since it required more than 5 seconds to execute.

N. Brisaboa, S. Ladra, and G. Navarro. 2014. Compact representation of Web graphs with extended functionality. *Information Systems* 39, 1 (2014), 152–174.

S. Büttcher, C. Clarke, and G. Cormack. 2010. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press.

D. R. Clark. 1996. *Compact Pat Trees*. Ph.D. Dissertation. Waterloo, Ont., Canada.

B. Croft, D. Metzler, and T. Strohman. 2009. *Search Engines: Information Retrieval in Practice*. Pearson Education.

J. S. Culpepper, M. Petri, and F. Scholer. 2012. Efficient in-memory top-k document retrieval. In *Proc. 35th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 225–234.

S. Culpepper, G. Navarro, S. Puglisi, and A. Turpin. 2010. Top-k ranked document search in general text databases. In *Proc. 18th Annual European Symposium on Algorithms (ESA B) (LNCS 6347)*. 194–205 (part II).

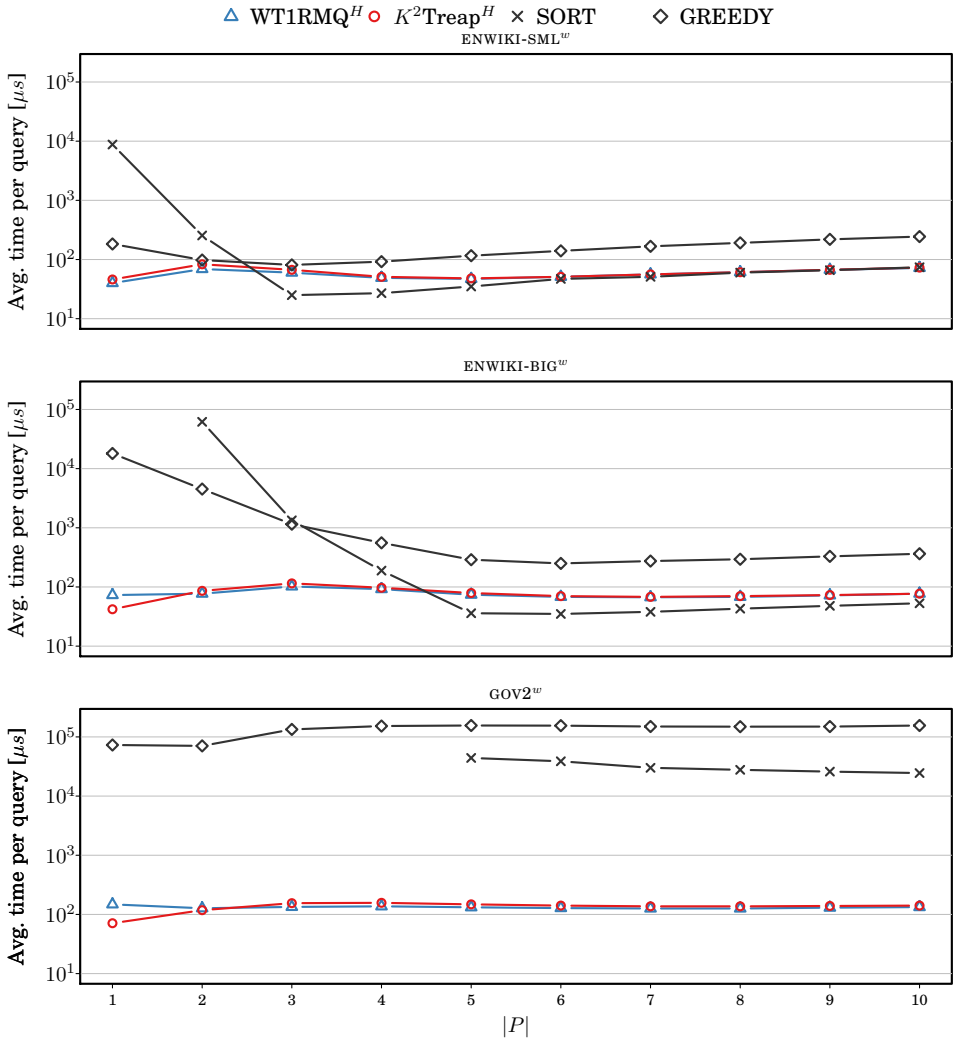


Fig. 26. Average time per query in microseconds for different pattern lengths. Results that required more than 5 seconds for SORT are not shown.

H. Ferrada and G. Navarro. 2014. Efficient compressed indexing for approximate top-k string retrieval. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE) (LNCS 8799)*. 18–30.

P. Ferragina and G. Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (2005), 552–581.

J. Fischer and V. Heun. 2011. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.* 40, 2 (2011), 465–492.

T. Gagie, G. Navarro, and S. J. Puglisi. 2012. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science* 426-427 (2012), 25–41.

S. Gog. 2011. *Compressed Suffix Trees: Design, Construction, and Applications*. Ph.D. Dissertation. Univ. of Ulm, Germany.

S. Gog, T. Beller, A. Moffat, and M. Petri. 2014. From theory to practice: Plug and play with succinct data structures. In *Proc 13th International Symposium on Experimental Algorithms (SEA)*. 326–337.

S. Gog and G. Navarro. 2015. Improved single-term top-k document retrieval. In *Proc. 17th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 24–32.

S. Gog and M. Petri. 2014. Optimized succinct data structures for massive data. *Software Prac. Experience* 44, 11 (2014), 1287–1314.

R. Grossi, A. Gupta, and J. Vitter. 2003. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 841–850.

W.-K. Hon, M. Patil, R. Shah, and S.-B. Wu. 2010. Efficient index for retrieving top-k most frequent documents. *Journal of Discrete Algorithms* 8, 4 (2010), 402–417.

W.-K. Hon, R. Shah, and J. S. Vitter. 2009. Space-efficient framework for top-k string retrieval problems. In *Proc. 50th Annual Symposium on Foundations of Computer Science (FOCS)*. 713–722.

J. Karkkainen, D. Kempa, and S.J. Puglisi. 2014. Hybrid Compression of Bitvectors for the FM-Index. In *Data Compression Conference (DCC), 2014*. 302–311.

R. Konow and G. Navarro. 2013. Faster compact top-k document retrieval. In *Proc. 23rd Data Compression Conference (DCC)*. 351–360.

N. J. Larsson and A. Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732.

T.-Y. Liu. 2009. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval* 3, 3 (2009), 225–331.

V. Mäkinen and G. Navarro. 2006. Position-restricted substring searching. In *Proc. 7th Latin American Theoretical Informatics (LATIN) (LNCS 3887)*. 703–714.

U. Manber and E. W. Myers. 1993. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.* 22, 5 (1993), 935–948.

I. Munro. 1996. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS) (LNCS 1180)*. 37–42.

J. I. Munro and V. Raman. 2002. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.* 31, 3 (2002), 762–776.

S. Muthukrishnan. 2002. Efficient algorithms for document retrieval problems. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 657–666.

G. Navarro. 2014. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *Comput. Surveys* 46, 4 (2014), article 52.

Gonzalo Navarro. 2016. *Compact Data Structures – A practical approach*. Cambridge University Press.

G. Navarro and V. Mäkinen. 2007. Compressed full-text indexes. *Comput. Surveys* 39, 1 (2007).

G. Navarro and Y. Nekrich. 2012. Top-k document retrieval in optimal time and linear space. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

G. Navarro, Y. Nekrich, and L. Russo. 2013. Space-efficient data-analysis queries on grids. *Theoretical Computer Science* 482 (2013), 60–72.

G. Navarro, S. J. Puglisi, and J. Sirén. 2014a. Document retrieval on repetitive collections. In *Proc. 22nd Annual European Symposium on Agorithms (ESA B) (LNCS 8737)*. 725–736.

G. Navarro, S. J. Puglisi, and D. Valenzuela. 2014b. General document retrieval in compact space. *ACM Journal of Experimental Algorithmics* 19, 2 (2014), article 3.

G. Navarro and K. Sadakane. 2014. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms* 10, 3 (2014), article 16.

D. Okanohara and K. Sadakane. 2007. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 60–70.

M. Patil, S. V. Thankachan, R. Shah, W.-K. Hon, J. S. Vitter, and S. Chandrasekaran. 2011. Inverted indexes for phrases and strings. In *Proc. 34th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 555–564.

R. Raman, V. Raman, and S. S. Rao. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms* 3, 4 (2007), article 43.

W. Szpankowski. 1993. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM J. Comput.* 22, 6 (1993), 1176–1198.

S. Vigna. 2008. Broadword implementation of rank/select queries. In *Proc. 7th International Workshop on Experimental Algorithms (WEA) (LNCS 5038)*. 154–168.

P. Weiner. 1973. Linear pattern matching algorithms. In *Proc. Switching and Automata Theory*. 1–11.

Received XXX; revised XXXXX; accepted XXXXX